# 致谢

本科生涯已进入尾声，回首四年大学生活，有太多的人给予过我帮助、支持，在此向你们致以最诚挚的感谢。

首先要感谢的是我的导师刘铁岩老师。我在微软亚洲研究院实习期间师从刘铁岩老师，刘老师在百忙之中抽出时间对我进行悉心指导，为我的工作提供宝贵的建议，向我推荐了大量的书籍与文章，慢慢教会我如何做研究。每次与刘老师的交谈都会让我受益匪浅，在这里再次表达我衷心的感谢。同时还要感谢的是我的直接导师王太峰师兄，太峰师兄在我实习期间对我的工作进行悉心指导，给予我研究的方向，耐心的纠正我犯下的错误以及在他的指导下完成了此次毕业设计的平台。感谢太峰师兄对我耐心的指导和教诲让我不断进步。

感谢我最亲密的良师益友兼战友戴涵俊同学。在我实习期间，涵俊同学给予了我莫大的帮助，从他身上我学到了编程的技巧，专业的知识，敏捷的思维与逻辑，认真负责的态度，为人处世的方式……我们在实习期间并肩完成了此次毕设所述的平台。

感谢于竟成同学与我们一同奋战完成此平台，感谢整个IECA组的其他组员给我的帮助和支持。在微软亚洲研究院实习期间，是你们带给了我最美好的回忆，我们已经融为一个大家庭。在这个毕业的季节，衷心的祝愿大家前程似锦。

感谢班主任谭立湘老师一直以来对我的信任和关心，感谢大学四年中每位任课老师对我的教诲，感谢我的室友张诰同学对我的照顾，感谢学习委员张义飞同学为班级做出的贡献和对我的帮助。

感谢大学四年里帮助过我的各位师兄师姐以及同学们，你们让我对科大有了强烈的荣誉感，让我免去了前进途中的彷徨。

特别感谢我的女朋友常雨菲同学，你在我大学四年中扮演最重要的角色，感谢你一直以来对我的体谅、信任和关怀，你一直是我奋斗的动力与源泉。

最后把我的感谢送给我的母亲，没有你就没有我今天的一切，你的健康与快乐是我最大的心愿。

目录

[致谢 1](#_Toc389592468)

[摘要 4](#_Toc389592469)

[Abstract 5](#_Toc389592470)

[第1章 绪论 6](#_Toc389592471)

[1.1 研究背景与相关工作 6](#_Toc389592472)

[1.2 主要工作概述 8](#_Toc389592473)

[第2章 大规模神经网络平台IECABrain概览 10](#_Toc389592474)

[2.1 动机 10](#_Toc389592475)

[2.1.1 现有问题 10](#_Toc389592476)

[2.1.2 需求 11](#_Toc389592477)

[2.2 平台架构 12](#_Toc389592478)

[2.2.1 架构预览 12](#_Toc389592479)

[2.2.2 核心设计 15](#_Toc389592480)

[2.3 平台特点 20](#_Toc389592481)

[第3章 表达式模板类矩阵库 22](#_Toc389592482)

[3.1动机 22](#_Toc389592483)

[3.1.1 现有矩阵库 22](#_Toc389592484)

[3.1.2 IECAMatrix特点 23](#_Toc389592485)

[3.2 数据结构 24](#_Toc389592486)

[3.3 模板框架 25](#_Toc389592487)

[3.3.1 表达式引擎 25](#_Toc389592488)

[3.3.2 表达式解析引擎 26](#_Toc389592489)

[3.3.3表达式执行引擎 27](#_Toc389592490)

[3.4 性能对比 28](#_Toc389592491)

[第4章 全连接网络的实现 30](#_Toc389592492)

[4.1全连接网络结构 30](#_Toc389592493)

[4.2 全连接网络训练算法 30](#_Toc389592494)

[第5章 word2vec的实现 34](#_Toc389592495)

[5.1 人工神经网络词汇表达模型 34](#_Toc389592496)

[5.2 word2vec的基本网络结构 34](#_Toc389592497)

[5.3 word2vec的实现——softmax近似 36](#_Toc389592498)

[5.3.1层次Softmax 37](#_Toc389592499)

[5.3.2 负样本采样法 38](#_Toc389592500)

[第6章 卷积神经网络的实现 40](#_Toc389592501)

[6.1 卷积神经网络的基本结构 40](#_Toc389592502)

[6.2 卷积神经网络的实现 41](#_Toc389592503)

[第7章 分布式网络的实现 44](#_Toc389592504)

[7.1 多模型并发 44](#_Toc389592505)

[7.1.1 模型拷贝 44](#_Toc389592506)

[7.1.2 无锁更新 45](#_Toc389592507)

[7.1.3 流式数据提供 46](#_Toc389592508)

[7.2 分布式训练 47](#_Toc389592509)

[7.2.1 数据分布式与模型分布式 47](#_Toc389592510)

[7.2.2 模型分布式的实现 49](#_Toc389592511)

[第8章 实验结果分析与对比 52](#_Toc389592512)

[8.1 MNIST标准数据集实验 52](#_Toc389592513)

[8.2 特殊网络实验 53](#_Toc389592514)

[8.2.1 卷积神经网络实验 53](#_Toc389592515)

[8.2.2 word2vec实验 54](#_Toc389592516)

[8.3 分布式实验 55](#_Toc389592517)

[8.3.1 MNIST数据集 55](#_Toc389592518)

[8.3.2 Text8数据集 56](#_Toc389592519)

[第9章 总结与展望 57](#_Toc389592520)

[9.1 文章小结 57](#_Toc389592521)

[9.2 未来工作展望 57](#_Toc389592522)

[参考文献 58](#_Toc389592523)

# 摘要

随着人工智能领域的发展与计算设备的快速更新，文本识别与图像识别的研究受到了越来越多的重视，而神经网络作为机器学习的一个有力工具，也越来越多的得到当代计算机科学家们的关注。

神经网络现有的开发工具很多，如Theano, Matlab Deeplearning Toolbox, caffe等，它们各自都有优缺点。本文希望设计并实现一种快速的，灵活的神经网络平台，使其可以支持分布式的网络结构以完成大规模的文本或图像学习任务。

首先本文设计并实现了名为IMatrix的矩阵库，其底层函数的调用使用MKL库以及CuBlas库进行加速，上层封装采用表达式模板类，支持CPU与GPU、稀疏矩阵与稠密矩阵的自由切换以及快速运算。

其次本文设计了通用的全连接神经网络平台，并在其上扩展有卷积神经网络、word2vec等功能。

最后本文实现了分布式网络的搭建，并完成大规模分布式的文本学习任务。

关键词：神经网络平台，矩阵库，快速，灵活，可扩展性强，卷积神经网络，word2vec，分布式。

# Abstract

With the rapid growth in the field of artificial intelligence and computational device, much attention has been paid in text and image recognition. As a powerful tool of machine learning, neural networks is followed with interest by more and more computer scientist.

There are many deep learning tools and libraries such as Theano, Deeplearning Toolbox, caffe etc. We hope to design and develop a fast, flexible neural networks platform, and make it supporting distributed networks structure for large scale text or image learning task.

First of all, we design and develop a matrix library called IMatrix, it calls MKL and CuBlas libraries when calculate matrix operation, and we encapsulate the library by a fancy method called expression template. It supports CPU and GPU, Sparse and Dense matrix computation.

And then we design the general full connection neural networks platform, and implement the convolutional networks and word2vec.

Finally, we implement distributed neural networks structure using multi server. It can support large scale text learning task.

Keywords: Neural Networks Platform, Matrix Library, Fast, Flexible, Scalable, Convolution Networks, Word2vec, Distributed.

1. 绪论
   1. 研究背景与相关工作

人工智能领域的研究一直具有十分重要的科学与社会意义。近年来，越来越多的学者致力于研究深度学习，并将其应用于语音识别、图像识别、自然语言处理等，并在各自的研究领域屡屡取得突破性进展。

深度学习是指利用计算机来创建深层次的人工神经网络计算模型，模拟生物神经网络的结构和功能，建立起复杂的模型进行学习与分析，从而令其可以按照人脑的机制识别图片、文字等信息。神经网络通常由大量的人工神经元联结构成，输入数据在其之间传导与计算，并且根据外界信息改变人工神经网络的内部结构，以此对输入和输出数据间复杂的关系进行建模。

1986年，Rumelhart和McCelland提出了误差反向传播算法[[[1]](#endnote-2)]，以此来对人工神经网络进行有监督的学习。自此，神经网络的研究与应用渐渐进入了人们的视野，一些科学家利用浅层的神经网络实现了汽车在公路上的自动驾驶[[[2]](#endnote-3)],之后又有科学家让计算机通过神经网络，学会了模仿人类的声音。慢慢的，人们发现人工神经网络具有一定的局限性，包括受限于当时硬件设备的计算能力，使训练神经网络耗时过长，并且不能处理规模过大的网络；同时网络加深层数后，由于当时神经元的激活函数普遍使用S型函数(Sigmoid)，误差在反向传播时随传播层数的增长呈指数级下降，所以通常多层神经网络只有最后几层学习到了有用的信息，导致其并未表现出比浅层神经网络更好的学习能力；另外，神经网络需要大量的数据进行学习，而当时的小数据集往往会出现过拟合现象，这些因素都导致了在整个90年代深度学习方面的研究进展很小。

1998年，Yann Lecun等人根据人眼感受野的工作原理，提出了首个进行了实际应用的卷积神经网络模型: LeNet-5[[[3]](#endnote-4)]。他将待识别的图片利用固定大小的卷积窗进行卷积以提取局部特征，并在卷积层后面添加了子采样层进行池化以增加神经网络对图像旋转、偏移的细微差别的鲁棒性。由于图片各个部位对应的卷积窗权值共享，所以极大的降低了参数的数量，加快训练速率，且不易发生过拟合。其卷积的特性在处理二维图像上对边缘信息特征的提取保证了其准确率。LeNet-5被广泛的应用在美国邮政编码与银行手写支票的识别。

2006年，Hinton等人为了克服深层网络误差传播的问题，提出了几种预训练的方法，包括受限玻耳兹曼机、栈式自编码、稀疏编码等方式[[[4]](#endnote-5)]，利用输入数据对神经网络的权值参数进行非监督的学习，使其预先学习到数据的分布信息，再利用标记数据进行监督学习以微调网络参数。利用这种方式训练出的多层人工神经网络具有非常好的表达。之后Hinton等人又提出了如Drop Out等在小数据集上抑制过拟合现象的方法[[[5]](#endnote-6)]，以及一些新的神经元激活函数如Rectified 函数[[[6]](#endnote-7)]等可以在误差反向传播时有效的传播梯度值。

在互联网时代，时刻都在产生大量的数据，而处理这些数据需要较高要求的硬件设备。近年来，图形处理单元(GPU)由于其优秀的计算能力以及并行度上的优势，已经逐渐代替CPU来进行大规模密集型科学计算。英伟达公司最新推出的Tesla K40 GPU，单精度浮点性能已达到4.29TFLOPS。Andrew Ng等人在论文中描述了他们将神经网络模型放到分布式GPU集群上以提高训练的迭代速度[[[7]](#endnote-8)]。至此深度学习的研究又进入了一个新的阶段。

随着人工神经网络的逐渐成熟，人们将其应用到图像识别、文本学习等领域，并取得了不错的成果。MNIST是一个手写数字数据集，其中包含共60,000个训练样本以及10,000个测试样本，每张图片大小为28\*28的0-255表示的灰度图。在2013年，Li Wan等人使用一种名为DropConnect的神经网络结构[[[8]](#endnote-9)]，在此数据集上达到了99.79%的分类正确率。CIFAR-10是一个彩色图片数据集，其中包含了10个物体种类共60,000张32\*32的RGB三色图片，其中50,000张训练样本, 10,000张测试样本。DropConnect神经网络结构，在此数据集上也可达到90.68%的分类正确率。Word2vec[[[9]](#endnote-10)]是谷歌公司发行的一个用神经网络计算连续分布式词汇表达的工具。用word2vec在英文维基数据集上进行训练，并在单词相似度推理任务中可以达到61%的正确率。

为了进行深度学习方面的研究，搭建一个快速、准确、灵活的神经网络平台至关重要。Theano[[[10]](#endnote-11)]是一个Python的机器学习库，用户可以使用它进行神经网络方面的研究与应用。支持GPU, 开发快捷是Theano的优点。Matlab拥有自己的Deep Learning Toolbox[[[11]](#endnote-12)]，其支持监督学习的前向传递、动态的网络等，同时也支持非监督学习的自组织映射等。使用这个工具箱，用户可以设计、训练神经网络，其最大的特点是具有可视化以及仿真功能。Cuda-convnet [[[12]](#endnote-13)]是Hinton等人开发的卷积神经网络 C++/CUDA实现。其主要用于GPU，用户可自定义自己的网络结构，它的优点是训练速度较快。Caffe[[[13]](#endnote-14)]平台与Cuda-convnet类似，也是用C++实现的卷积神经网络，不过其在CPU上也有很好的表现。Accord.NET[[[14]](#endnote-15)]是用C#实现的机器学习库，它支持深度置信网以及受限玻耳兹曼机。

* 1. 主要工作概述

本文的工作主要包括以下几个方面：

1. 表达式模板矩阵库：

利用表达式模板的方式，封装了CPU与GPU的矩阵运算库。在编译的时候解析表达式，并直接展开成对应矩阵运算函数，避免了开辟和使用临时变量，提高了效率。

1. 全连接网络：

为了使用户便于搭建自己的神经网络结构，将神经网络模型抽象成层(layer)与连接(connection)类的组成。其中，层的子类包括输入层、隐层与输出层；对于全连接网络，连接的子类包括全连接与稀疏连接。由层与连接构成的有向无环图形成一个完整的神经网络。

1. word2vec网络:

word2vec网络在输出层采用哈夫曼编码或负采样(negative sampling) 。

1. 卷积神经网络:

由于卷积神经网络在层与层之间进行卷积与池化的操作，故为连接添加子类卷积连接与池化连接。

1. 分布式网络：

利用消息传递接口(MPI)可以在服务器间进行进程级别的通信。若神经网络的规模较大，可以将模型切分并放置在多台服务器上；若训练数据量较多，可以将完整模型放在多台服务器上，对每个模型分别利用部分数据训练，若干次迭代后进行模型的平均。分布式的网络可以处理大规模的网络和数据，并且可以加速训练的过程。

本文的组织结构如下：

* 第二章对本文所述平台进行概览性的探讨，主要叙述其动机，主要架构及特点。
* 第三章介绍了表达式模板矩阵库的实现。主要介绍了CPU与GPU的矩阵运算库，以及运用表达式模板的方法进行封装。
* 第四章主要介绍了全连接网络的实现。其中包括层与连接的实现，以及其各子类的实现。
* 第五章介绍了word2vec网络的结构与实现。
* 第六章简要介绍了卷积神经网络的实现，包括卷积连接与池化连接等。
* 第七章介绍了分布式网络的实现。包括多机间的通信与网络模型的切分等。
* 第八章分析了我们的实验结果，将其与现有的平台进行了对比。
* 第九章进行了简要的总结，并对以后工作进行了展望。

第2章 大规模神经网络平台IECABrain概览

这一章主要介绍大规模神经网络平台的高层次信息。首先将阐述我们构建IECABrain的动机，通过对现今学术界和工业界的神经网络平台进行回顾，我们会发现他们具有各种不同的问题，而无法满足我们的需求；接下来，将概括性介绍我们开发的神经网络平台的设计架构和实线理念，具体的实现方法和算法将留在后面的章节中具体阐述；最后，我们总结新构建的平台的特点。

## 2.1 动机

构建一个通用的神经网络平台是一件需要大量精力与时间的工作。为了能够支持我们后续的研究工作，我们有必要构建适合自己的神经网络平台——IECABrain。以下将通过横向对比现有的神经网络平台、以及结合自身研究课题需求，阐述IECABrain诞生的意义，以及其所要完成的任务。

### 2.1.1 现有问题

在当今的学术界和工业界，大数据和机器学习已经成为了热门关键词。而大数据带来的不仅仅是机遇，还有对现有机器学习算法以及软件开发人员的巨大挑战。神经网络在最近兴起的深度学习中，又再次引起了一波研究热潮。不过，由于多层神经网络的训练并不是一个凸优化问题，因而要训练好一个神经网络，即防止得到较差的局部最优解，需要很多的技巧与经验。

通过上文中对现有的神经网络平台的介绍，我们对其特点有了初步的掌握。在开发我们自己的平台之前，我们对这些现有的平台进行了详细的调查。通过安装并运行一些标准数据集，我们发现这些平台有如下的一些问题：

1. **Matlab[[[15]](#endnote-16)]**

Matlab一个主要的问题是非免费。它是个收费软件，因而要部署在集群上，就需要有多个许可证，而其收费也是非常高昂。Matlab程序编写容易所带来的代价便是效率的降低，尤其是和优化过的C++代码的效率相比。另外，在Matlab的神经网络工具包中，其包装的完整性也使得我们并不方便进行深层次的定制，而用其进行神经网络的研究，必须要研究、改变其底层结构和实现。这些原因使得Matlab不满足我们的研究需求。

1. **Theano**

虽然Theano可以编译成C代码执行，也可以方便地利用GPU运算，不过效率上还是不如优化过的C代码。另外，Theano更多的是可以看做一个符号运算的执行器，相对于神经网络来说，负责的是更底层的运算。考虑到后续开发和维护的代价，用高效的C++更为合适。

1. **Pybrain[[[16]](#endnote-17)]**

该项目已经多年没有更新了。虽然是一个完整的神经网络库，并且在设计方面有其独到之处，但是效率问题（并不支持MKL以及CUDA优化）以及扩展能力的不足（不能进行分布式运算），使得其离我们的需求还很远。

1. **FANN[[[17]](#endnote-18)]**

该平台虽然完全用C代码编写，但是由于没有用并行优化，也难以扩展到GPU平台和分布式框架，所以更多的适合于在单节点上小数据集的调试。另外，神经网络训练算法也固定在其设计中，导致批量梯度下降、随机梯度下降等方法不适用。

1. **OpenCV[[[18]](#endnote-19)]**

OpenCV使用的人较多，便于交流，速度快，可以跨平台使用，但目前尚不支持卷积神经网络的部分代码。且OpenCV最大的问题在于版本不大稳定，且Bug较多。

1. **Caffe**

该平台更多的偏向于使用，而并不是二次开发。一方面，它并不支持分布式的神经网络训练；另外，虽然在Caffe平台下配置一个神经网络，只需要写少量的标记语言，但是，这也限制了用户能够修改的部分。对比我们对平台的要求：适于应对研究工作中各种修改、有分布式计算的能力，Caffe显然也不是一个合适我们当前任务的选择。

### 2.1.2 需求

为了支持神经网络相关的研究工作，一个强大而高效的神经网络平台是必须的。具体而言，我们当前的任务主要集中在大规模文本处理类应用如计算广告学等方面。

作为新兴的分支学科，计算广告学涉及到了大量研究内容，如机器学习、信息检索等。而作为一个典型的计算广告学的应用，搜索广告占有了重要的地位。通过在搜索引擎中投放广告，进而通过用户的点击进行收费，这种模式已经成为了大量的互联网公司的重要盈利手段。而广告平台对广告主收费的重要依据就是广告的点击，因此，投放用户最可能点击的广告，是广告投放平台和广告主都关心的问题。而广告点击预测最为显著的特点，就是——数据量巨大。平均而言，主流的搜索引擎（如Google，百度，Bing等）每日的搜索量都在10亿级别。每次用户的搜索，系统都会投放一定数量的广告。如果我们将过去一个月的用户搜索广告点击行为数据收集起来，用机器学习的方法去训练，将是对所用的算法以及系统一个巨大的挑战。

另外，用户的行为也在随着时间变化。而且广告点击的行为不仅仅与用户相关，还与当前的热点事件、节假日等有密切关系。因此，训练出来的模型也需要经常更新。一般可以用在线学习的方式，也可以每两到三天更新一次。

我们不难从上面总结出，在这个问题中，如果用神经网络模型作为机器学习的工具，那么就需要训练网络的平台具有如下的特点：

1. 平台具有很好的扩展能力。这种扩展能力主要体现在当计算资源变多了之后，我们能够有办法利用大量的服务器去提升计算能力，从而在短时间内训练模型，以满足计算广告的问题中关于时效性的需求。因而，该平台能够有分布式计算的能力。
2. 在计算广告问题中，常常会需要用bag-of-words形式的向量，即所谓的ID特征。考虑到作为特征的对象的规模常常非常大，例如广告的数量、用户的数量。这些将导致一个问题——由于单个节点的内存大小有限，我们的模型甚至无法在单个运算节点（服务器）上面存下来！所以，平台需要有能力将模型的参数也分布式存储。

## 2.2 平台架构

在这一小节，我们将首先从大规模神经网络平台IECABrain的层次架构开始，介绍平台中各个部分的依赖关系；接下来会按照层次，从底层向高层介绍IECABrain，首先我们用一个小节介绍我们自主开发的矩阵运算库，这个矩阵库支撑了整个平台的核心运算，接着，便是核心部分的设计介绍，这包括了网络核心的各个部件，以及其组织逻辑。

### 2.2.1 架构预览

像众多软件的设计方法一样，IECABrain也采用多层次架构进行设计。具体来说，我们将架构分为五层，如图2.1所示。

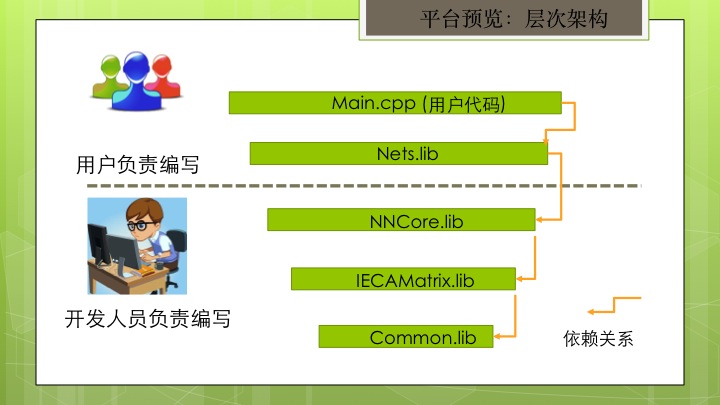


图2.1 平台架构预览

可以看到，为了最大限度地让我们的平台易于扩展，需要用户编写少量的代码——也就是说，我们的平台不是一个傻瓜式的黑盒，只需要输入参数和数据即可，这不是我们的初衷。我们的目的是构建一个适合进行关于神经网络和深度学习的深层次研究的平台。而这就需要平台易于扩展，同时也要易于维护。

除了最顶层的用户代码，其余几层都被编译成了静态链接库。相较于将所有层次构建在一个项目编译输出，这种设计能够减少开发过程中编译的代价。同时，也易于之后的维护。

下面，我们由底层到顶层，分别介绍每个层次所负责的功能。

1. **Common.lib**

Common是通用的库，顾名思义，这个是最底层的，与神经网络几乎无关的库。该库主要包含许多的工具类，后文中将会直接使用到其中的一些类，由于其实现方法不是本文讨论的重点，所以统一在这里进行列举说明。主要用到的类有：

1. **Dictionary**

一种按照键-值对组织的数据结构。通过可以计算哈希编码的键来索引对应的值。核心部分是一个哈希表。这种实现方式平均情况下会比平衡树来的更高效，当然，是以一定空间代价作为交换的。

1. **HuffmanEncoder**

一个自动进行哈弗曼编码的工具。通过给定每个编码对象的字符串，以及该对象的权重（譬如单词的词频），该工具能够自动构建哈弗曼树，同时能够知道每个编码对象的编码、其在哈弗曼树上的路径等重要信息。

1. **CMultinomialSampler**

一个随机采样工具。该工具能够进行多项分布的采样。通过给定初始的项数和每项对应的概率，该工具能够很高效地对给定的多项分布进行采样，同时，其基于c++的伪随机数，方便调试。

1. **Log**

一个记录日志的工具。相比较工业界流行的glog之类的工具，这个工具更加轻量级，并且易于部署、管理。

更多的类和每个类各自的实现方式，请参考源代码。在后文中，这一个库就不会再进行更多的介绍。

1. **IECAMatrix.lib**

神经网络中，绝大多数操作都可以转化成矩阵运算。例如，两层网络之间的前向传播，就可以用一个矩阵乘法来表达。将这层运算用矩阵封装后，在更高层次，我们就只需要关注神经网络算法的逻辑，而不需要花费更多精力在具体运算实现上面了。

因此，我们有必要建立一个矩阵库，专门负责这些操作。IECAMatrix就是这样的一个通用矩阵库。之所以称之为通用，是因为它不仅可以服务于IECABrain这个神经网络平台，还可以用于其他的项目中。

然而，这个矩阵库有两个最主要的特点：统一性和优雅性。我们将在下一章节，**IECAMatrix矩阵库**中，进一步介绍。

1. **NNCore.lib**

该库包含了我们神经网络平台的核心部件。在设计的逻辑中，两个最主要的类——层（Layer）和连接（Connection）将在这一个层面实现。

除了构成神经网络的层和连接这些部件之外，还有配套的许多其他部件，例如流式数据读取、分布式计算、多线程并行，以及大规模神经网络训练的优化方法等，都会在这一层面实现。因而，这个库是整个神经网络平台的核心。

当然，这里不仅仅有一系列单独的部件，还涉及到这些部件如何协同工作的逻辑。这些内容将在下面的章节**核心设计**中进一步介绍。

1. **Nets.lib**

从这里开始，就要涉及到用户编写的代码了。有了NNCore这个核心部件，构建各种神经网络，就只需要组装这些部件了。而不同的神经网络的差别，大多也是在结构上、层和连接表现行为上的差异，因此，留给用户写的代码并不是很多。在**核心设计**这一节中，也会展示一个通用的抽象神经网络，即，通过配置不同的部件，同一个神经网络类可以展现完全不同的功能。

1. **Main.cpp**

这里完全是用户自己编写的代码。主要涉及到用户的个性化配置，例如矩阵运算用CPU还是GPU实现，数据文件存放在哪里，训练神经网络的时候，各个可以调控的参数该如何设置，等等。

至此，我们按照层次结构从底层到高层，完整介绍了我们的平台概况。

### 2.2.2 核心设计

在这一节，我们主要介绍上文中提到的NNCore.lib，也就是神经网络平台的第三层。通过对各个部件的功能以及对其组织方式进行介绍，我们将从整体上把握整个平台的设计理念。在之后的章节中，我们会详细说明各个部件的实现方法，以及在各种标准数据集上，与现有平台的性能对比。

我们首先分门别类介绍各个部件。接下来，我们按照各个部分负责的逻辑上的功能，用示意图结合文字的形式，介绍它们组合运作的方式。

#### 2.2.2.1 层与连接

神经网络结构中，层和连接是最关键的两个组成元素。如图2.2展示了一个普通的三层全连接神经网络，在此图上，我们定义了层和连接这两个概念。

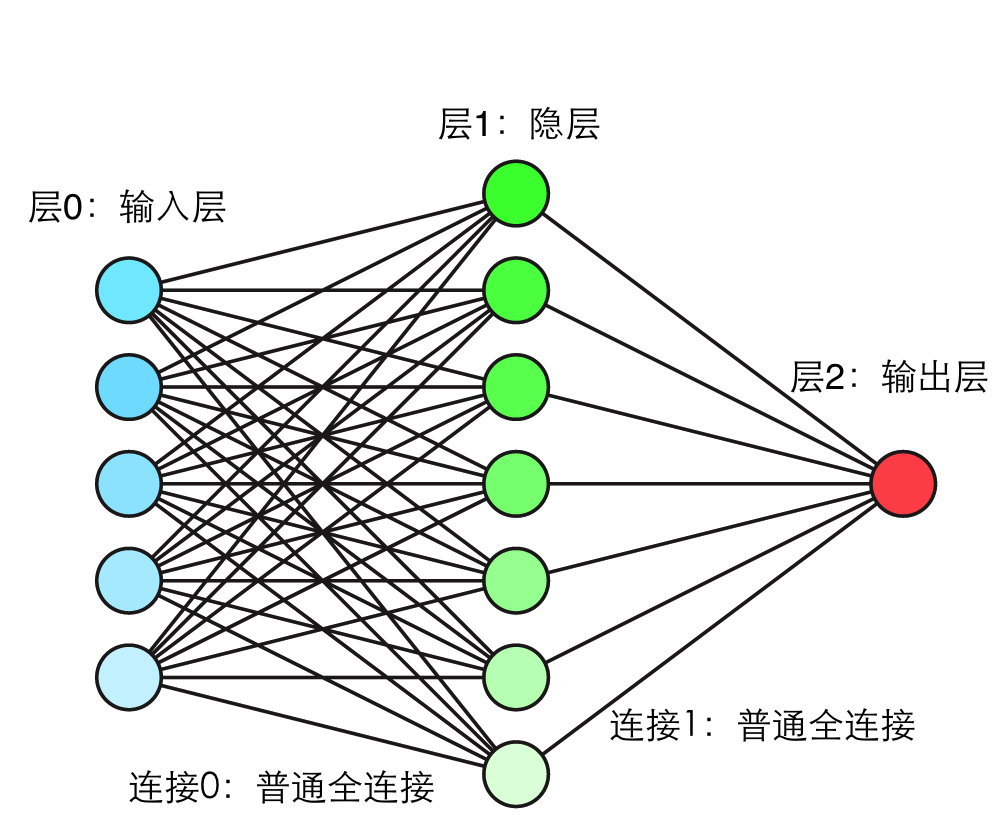


图2.2 三层全连接神经网络

图2.2中，一共有三层（层0~层2），两个连接（连接0和连接1）。因此，在我们的定义中，一层对应了一个神经元的集合，而一个连接，对应了连接两层之间的边。以上是数据结构层面的定义，而神经元和边连接又可以有多种表现行为，比如神经元的激活方式、边的前向传播方式以及反向更新方式等，都可以有变化。因此，我们将这两个元素——层和连接，用以下的数据结构来表示。

* **层**

如上所述，层是一个神经元集合的组织方式。而层的标号（0，1，2，等等）并不一定需要有顺序关系（即层1不一定代表着在层2之前），而仅仅是用来作为层的一个名字，用来唯一标识层。这种定义方式给我们平台带来了很大的灵活性——我们可以自己定义层的拓扑顺序。

图2.3是核心部件——层的类图。下面选取主要的类介绍。

* + **AbstractLayer**

所有层的抽象父类。定义了层所拥有的基本数据成员，以及主要的前向和反向传播的运算逻辑。这里有必要介绍一下其所拥有的基本数据成员。

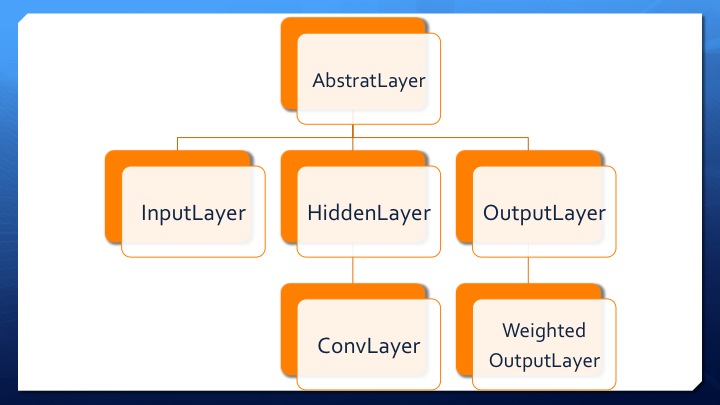


图2.3 层的类图

可以看到，层的数据用矩阵IMatrix来存储。

template<MatMode mode>

struct AbstractLayer

{

public:

IMatrix<mode>\* activations; //存放激活值

IMatrix<mode>\* errors; //存放反向传播误差

unsigned uThreadReplicaID; //多线程的ID

std::vector<Edge> inConns, outConns; //连接集合

const AbstractLayerInfo<mode>\* pLayerInfo;

//层的配置信息，如层的大小、编号，等等。

};

* + **InputLayer**

输入层负责将数据读入到当前的神经元中，作为输入层的激活值，并用来做前向传播。由于输入层不需要更新，因而不需要做反向传播。和隐层以及输出层最大的区别是，其神经元的激活值是从数据中读取而来的。

* + **HiddenLayer**

神经网络的隐层。负责前向传播和反向传播。

* + **OutputLayer**

神经网络的输出层。除了负责反向传播之外，还需要负责载入监督信息，以计算当前的输出值与标准值的偏差，用来反向传播误差。

* **连接**

故名意思，连接即是在两个层之间存在的边。连接可以有多种方式，比如全连接（Full Connection），也可以局部连接（Local Connection），或者相对于普通的前向传播，可以有随机丢失连接（Drop Connection），可以用卷积连接，等等。每种连接有着自己的特点，当然，他们也有着共同的地方，所以和层一样，我们也用子类继承方式组织这些不同的连接。另外，连接的编号也只是用来唯一区别，并不代表顺序或者其它一些信息。

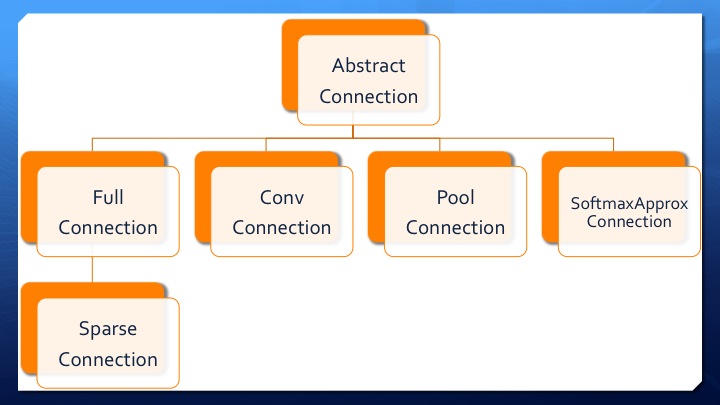


图2.4 连接的类图

图2.4是连接的类图。这里只介绍最主要的抽象类。

* + AbstractConnection

是所有连接的抽象父类。同样包含了基本的数据成员。如下所示。同样，连接所对应的边用矩阵来存储。

template<MatMode mode>

struct AbstractConnection

{

public:

IMatrix<mode>\* pWeights; //连接的边权

IMatrix<mode>\* pBias; //神经网络的偏置项

unsigned uConnID; //连接的编号

BiasStatus biasStatus; //是否需要使用偏置

MatFormat format; //连接是稀疏还是稠密

};

其它一些类将在后文中具体介绍。

#### 2.2.2.2 有向无环图结构

IECABrain平台的一大特点，就是可以训练任何结构合法的神经网络。我们对结构合法的定义如下：用图论的观点，如果把层看做图中节点，把网络连接看做图中连接两个节点的有向边，其中边的方向就是神经网络前向传播的方向，那么，该图是个有向无环图（DAG）。

如果把神经网络看做图之后，图中存在环的话，那么这种训练行为是未定义的，必须得有额外的限制条件。所以，IECABrain在这一方面，已经是最大程度地支持灵活的网络结构了。

维护该图结构的是CDAGStructure这个类。该类的主要方法和数据结构如下：

template<MatMode mode>

class CDAGStructure

{

public:

//判断网络结构是否合法，并保存经过拓扑排序的层和连接

bool IsStructureValid();

//添加新的层

bool AddLayer(AbstractLayerInfo<mode>\* pLayerInfo);

//给两个层之间添加新的连接

bool AddConnection(unsigned uLayerFrom, unsigned uLayerTo, AbstractConnection<mode>\* pconn);

// 保存添加的层的信息

std::vector<AbstractLayerInfo<mode>\*> m\_layerinfo;

// 将层按照拓扑顺序存放

std::vector<AbstractLayerInfo<mode>\*> m\_layerInOrder;

};

从上面的结构可以看出，为了构造一个神经网络，我们将会像构建一个DAG一样，添加层和连接。经过拓扑排序，我们除了能够验证网络结构的合法性，还能得到按照拓扑顺序排列的层。不难想到，如果按照拓扑顺序对这些层进行前向传播以及逆拓扑序进行反向传播，我们就可以像训练普通的三层神经网络一样，训练一个多层的符合拓扑顺序的网络。

#### 2.2.2.3 流式数据提供

要让神经网络的训练过程收敛到较好的解，大量的训练数据是必不可少的。而数据量大了之后，我们便不可能将所有的数据都存放在内存中直接使用了，因此，我们设计了流式的数据提供方式——在这种方式下，只要时间允许，就可以训练无穷数量的数据。

这部分内容将会涉及到部分优化算法的问题，具体内容将在1.3节介绍。神经网络常常使用批量梯度下降进行训练。每次，一批数据会被读入内存，通过前向传播和反向传播对神经网络进行更新，接着下一批数据被读入。如此反复直到将所有的数据都通过神经网络，一个周期的训练便结束了。

用来提供流式数据的类是CDataSupplier。该类不负责读入数据，而只是将读入的数据进行缓存。输入层通过访问CDataSupplier来获取当前批次的数据。有了这个结构，在分布式框架和多模型训练情况下，输入层不需要考虑数据同步等问题，直接交由CDataSupplier即可。

而具体负责读入数据的，是DataReader所负责的。由于数据格式多种多样，为了使得能够适配我们的平台，需要有相应的数据适配器，这就是DataReader所负责的事情——把不同格式的数据都整理成神经网络能够接受的矩阵形式。

首先，如图2.5所示DataReader的组织结构以及对应接口的用法。

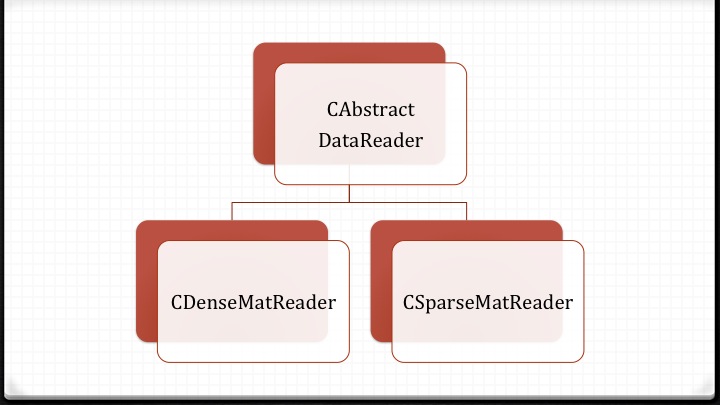


图2.5 DataReader的类图

其中，CAbstractDataReader规定了所有数据适配器（Reader）必须实现的接口，以及一些共有的数据成员。两个具有代表性的子类CDenseMatReader和CSparseMatReader分别是读入稠密矩阵和稀疏矩阵的适配器。用户可以遵循这一原则，自主开发数据适配器来匹配自定义格式的数据。

## 2.3 平台特点

IECABrain总结了现有的平台的优点，以及最新的学术界研究成果。它在如下几个方面具有突出的特点：

* **效率**

IECABrain完全使用高效的C++编写，其计算效率和算法效率都进行了深层次的优化。

* + **矩阵计算**

底层基于自主开发的IECAMatrix矩阵库。该矩阵库可以分别在CPU和GPU平台使用。在CPU平台利用MKL提供的针对Intel指令集和多核心优化，在GPU平台则依赖CUDA提供的并行运算，同时加上特有的延迟计算优化，使得矩阵库本身就比其它一些常见的矩阵库要高效。

* + **算法优化**

一方面，通过softmax近似算法，大大减少了分类问题中输出层计算代价；另外一方面，利用多模型并行训练，使得训练周期缩短。

* **能力**

IECABrain能够同时处理大数据和大模型，具体表现在：

* + **流式数据训练**

通过流式数据和批量梯度下降，IECABrain能够训练海量的数据，而仅需基本的内存需求。

* + **分布式计算**

通过将模型分布式存储，只要有足够的节点，IECABrain就可以训练任意规模的神经网络。

* **易用性**

在底层运算方面，IECAMatrix使用运算符重载，以最自然的算术表达式形式执行矩阵运算。

在神经网络层面，用户只需要像搭积木一般，将层和连接按照自己想要的方式组合起来，设置好需要的训练参数，即可以使用强大的IECABrain。

* **扩展性**

IECABrain本身的初衷便是成为一个适合研究和拓展的神经网络平台。平台中的每个部件几乎都可以由用户自己进行扩展。从底层的数据接口，到核心部分的层和连接的表现行为，再到高层的神经网络构建和训练方式，都可以按照约定的接口实现自己所需要的功能。

# 第3章 表达式模板类矩阵库

为了支持IECABrain的底层运算，我们构建了自己的矩阵库——IECAMatrix。该矩阵库是通用的用于矩阵算术运算的库。首先，我们将从动机出发，阐述构建该库的原因，以及其所具有的特性；然后，我们从实现角度，介绍IECAMatrix的数据结构，以及其特性之一——C++模板框架。利用该特性，我们就能用少量的代码，实现符合自然语言习惯的矩阵算术运算表达。最后，为了验证其高效性和有效性，我们用常用的矩阵乘法，对比IECAMatrix和其他一些平台的性能。

## 3.1动机

IECAMatrix是一个通用的矩阵运算库。其同时支持稠密矩阵、稀疏矩阵和稠密矩阵与稀疏矩阵之间的运算。另外，该库可以在CPU平台或者GPU平台运行。这些功能满足了依赖该矩阵库的IECABrain的各个需求：1）CPU与GPU平台通用；2）同时支持稠密/稀疏特征，以及稠密/稀疏网络连接。可以说，IECABrain的灵活特性，很大一部分依赖于IECAMatrix的灵活。

下面，我们首先对现有的流行的矩阵库进行介绍；接着，通过进一步介绍IECAMatrix，以及对比现有的矩阵库，将阐明构建IECAMatrix的动机。

### 3.1.1 现有矩阵库

这一节，我们将介绍工业界和学术界常用的几个矩阵运算库的特点，同时对比各自的优缺点。

* **Matlab**

Matlab中同时支持稠密矩阵和稀疏矩阵，以及两者之间的转换和相互运算。另外，Matlab也可以同时在CPU平台和GPU平台进行运算，而且其矩阵算术运算的表达也符合自然语言习惯。但是，相比较高效的C++代码实现，Matlab的矩阵库就稍逊一筹。另外，如果混用C++和Matlab代码，中间的程序接口也会需要一定代价。

* **OpenCV**

OpenCV拥有自己的矩阵运算库。该库同样支持稠密矩阵的运算，并且在CPU上能够使用MKL加速。但是，在GPU上的运算接口与CPU中的不同，同时，稀疏矩阵并不像稠密矩阵那样获得足够的支持。实现的效率不够高也是缺点之一。

* **Eigen**

Eigen[[[19]](#endnote-20)]是一个开源的矩阵库。除了能够进行基本的矩阵算术运算之外，还实现了矩阵分解等算法。Eigen能够在CPU平台运行，并能够使用Intel的MKL加速，用C++实现的模板类也非常高效。不过，它并不支持GPU运算。同时，虽然Eigen同时支持稀疏矩阵和稠密矩阵，不过这两个矩阵之间并不方便进行抽象，如果使用Eigen，这个问题会给IECABrain的开发带来一定困难。

在后面的小节中，我们将把IECAMatrix和这几个矩阵库放在一起进行性能测试，通过对比来证明IECAMatrix的性能。

### 3.1.2 IECAMatrix特点

IECAMatrix具有的特点如下：

* **统一**

在矩阵运算中，我们统一表达了CPU/GPU上的运算，以及稠密/稀疏矩阵的存储。

具体而言，IECAMatrix记录了模板变量MatMode，即矩阵的运算模式。模式可以有CPU运算模式和GPU运算模式。另外，通过C++的多态，我们能够用统一的接口访问具体的矩阵实例——即稠密矩阵或者稀疏矩阵。

通过这种方式，我们可以在仅修改配置参数的情况下，让IECAMatrix应对不同类型的计算。

* **优雅**

在矩阵算术运算方面，我们重载了C++的运算符，使得矩阵算术运算能够像一般变量的算术运算一样书写。通过重载C++的“+”，“-”，“\*”和“/”运算符，我们实现了矩阵的加法、减法、乘法、数乘，以及矩阵和矩阵之间逐元素的运算。

#define MODE MatMode::CPU

auto a = DenseMat<MODE>(10, 10);

auto b = SparseMat<MODE>(10, 5);

auto c = DenseMat<MODE>(5, 10);

a -= b \* c \* 0.001f;

以下的一段代码，实现了矩阵的乘法、数乘和减法。

* **高效**

在CPU平台，我们采用Intel的MKL运算库支持；而在GPU平台，使用CuBlas实现底层的矩阵运算。另外，由于使用了表达式模板引擎，使得我们能够对表达式本身进行优化，一方面减少临时变量的开销，另外一方面也减少了冗余运算。

## 3.2 数据结构

为了统一稠密矩阵和稀疏矩阵，我们用了一个抽象矩阵类IMatrix，它的两个子类，分别为SparseMat（稀疏矩阵）和DenseMat（稠密矩阵）。

通过IMatrix的data域，我们可以访问到具体子类的数据结构。

对于稠密矩阵，我们存储的数据结构如下：

template<MatMode mode>

struct DenseData

{

public:

int rows, cols;

matEleType\* pDense;

};

其中matEleType是预先定义的浮点数类型，可以是单精度，也可以是双精度。可以看到，实际存储二维矩阵元素的，是一维数组pDense。

对于稀疏矩阵，我们存储的数据结构如下：

template<MatMode mode>

struct SparseData

{

public:

matEleType\* pVal; //非零值

int\* p1BasedRowIdx; //非零元素行号

int\* p1BasedColIdx; //非零元素列号

int\* p1BasedCSCColPtr; //CSC格式列指针

int nnz; //非零元素个数

int ncol; //CSC格式列数

int nzCap; //非零元素个数上限

int colPtrCap; //CSC格式列数上限

};

这里，我们使用CSC的稀疏矩阵存储格式。同时，为了访问方便，也采用COO的存储方式。这些均是通用的稀疏矩阵存储格式，具体可以参考相关文档。

用户可以直接对这些域进行修改，也可以使用预先编写的常用函数，获取所需的矩阵运算功能。

## 3.3 模板框架

为了方便地实现符合自然语言习惯的矩阵算术运算表达，IECAMatrix采用了C++的模板。通过用模板抽象出表达式，再利用模板匹配，求解对应的具体表达式，一个从抽象到具体的自上而下的设计框架就呈现了出来。

整个模板框架自上而下分为：1）表达式引擎，负责表达式的构建；2）表达式解析引擎，负责解析构建好的表达式，调用具体的运算执行引擎；3）表达式执行引擎，对具体的运算表达式，判定运算操作数和操作符，并调用相应的库函数（MKL或者CuBlas，分别对应CPU和GPU的运算）。

### 3.3.1 表达式引擎

表达式引擎，其主要职责是负责构建表达式树。具体而言，表达式引擎将一个完整的表达式按照其执行顺序（优先级）组装成一棵表达式树，再将该树交由表达式解析引擎进行解析。

在一个表达式中，有很多的子表达式。为了方便之后的解析，每个子表达式应该足够纯粹，使得每个子表达式都能用单一的解析引擎解析，而不需要有过多的依赖关系。如图3.1所示，是属于表达式引擎的各个元素之间的对应关系。

图3.1 表达式引擎

* **Expr**

表达式的抽象父类。该父类只有两个成员变量，即rows（行数）和cols（列数）。而Expr类最重要的两个函数，就是得到其衍生类的方法。

每个Expr的子类，会将自身的类型作为模板参数。这样，在仅知道抽象类的情况下，也能获得具体子类的实例。

template<typename DerivedType>

struct Expr

{

public:

int rows, cols;

inline DerivedType& Derived()

{

return \*static\_cast<DerivedType\*>(this);

}

};

* **LvalueExpr**

即左值表达式类型。该类型的表达式可以放在赋值符号的左边。典型的这种表达式类型有矩阵和子矩阵。

* **MultiplyExpr**

矩阵乘法表达式。这种表达式区别于矩阵加法和减法表达式。

* **BinaryCwiseExpr**

矩阵和矩阵之间逐元素的运算表达式。包括矩阵加法、减法和数乘。

通过重载运算符，我们可以根据运算符以及运算符两边的操作数，决定构建的子表达式类型。

### 3.3.2 表达式解析引擎

在3.3.1节，我们介绍了如何构建表达式树。因此，真正的表达式计算，并不是在构建表达式树的过程中进行的，而是在赋值符号解析的时候。

解析表达式树的目的，是将封装好的表达式树按照运算优先级关系，将每个子表达式交予对应的表达式执行引擎运算，同时也为了便于充分对整棵表达式树进行运算和表达的优化。

表达式解析引擎分为三类：

* **UnaryCwiseEngine**

解析单元表达式。例如，A += B，A \*= 2.0。这种表达式的右边只有一个操作数，因此可以针对这种表达式做特殊的优化。

* **CwiseExprEngine**

解析二元逐元素运算表达式，例如A = B + C。这种情况下，我们可以直接将B + C的结果存入A，而不用先新建临时变量存储B + C的结果，再将结果赋值给予C。

* **CmplxExprEngine**

解析复杂表达式。例如矩阵的转置，以及矩阵乘法。值得注意的是，除非矩阵的转置会被赋予到新的矩阵中，否则，该转置操作不会被真正执行，只需要记录标记即可。这样可以免除新建临时变量，并减少不必要的操作。

### 3.3.3表达式执行引擎

表达式解析完成后，便交由表达式执行引擎进行计算。在这一层面，我们才会需要区分矩阵的类型，即稠密矩阵/稀疏矩阵，以及其所使用的模式，包括CPU和GPU两种。

#### 3.3.3.1 CPU引擎

Intel® Math Kernel Library [[[20]](#endnote-21)]是英特尔公司发行的科学计算库。其提高了软件应用程序在进行大规模数学运算时的效率。英特尔MKL库包含了基础线性代数程序集(BLAS, Basic Linear Algebra Subprograms)、Fortran线性代数包(LAPACK, Linear Algebra PACKage)，快速傅里叶变换，量化数学函数以及随机数生成等接口。在本文所述平台中，主要在CPU模式下运用英特尔MKL库的基础线性代数程序集以及向量数学函数进行矩阵运算。

#### 3.3.3.2 GPU引擎

CUDA™ 是 NVIDIA® 公司的并行计算架构。The NVIDIA® CUDA® Toolkit [[[21]](#endnote-22)]是其公司发行的基于CUDA架构的开发包，它为开发利用GPU加速的程序的开发者们提供了一个详尽的基于C/C++的开发环境。CUDA Toolkit包括NVIDIA GPU 的编译器，数学函数库以及一些用于调试和优化程序的工具。

在本文所述平台中，主要在GPU模式下运用NVIDIA CUDA Toolkit中的cuBlas库进行稠密矩阵的运算，cuSparse库进行稀疏矩阵的运算以及利用cuRand库作为GPU上的随机数生成器。

## 3.4 性能对比

下面将通过两个简单的矩阵算术运算实验，对比IECAMatrix和一些主流的矩阵运算库，分别是：Matlab，OpenCV，Eigen。

实验硬件环境：Windows 7操作系统，4G内存，CPU为Intel Core i5-2410M双核，主频2.3GHz每个核。

我们将执行如下的矩阵运算：

A += B \* C

其中A、B、C均为N \* N的单精度浮点数矩阵。上面的运算同时涉及到矩阵的乘法和加法。我们按照[-0.1，0.1]的均匀分布随机初始化矩阵B和矩阵C（当然，在每个库的实验时，我们保证B和C按照同样的随机种子进行随机，以使得每组实验数据完全相同）。

* **小规模矩阵测试**

首先，我们将矩阵的大小N设置为100。我们将上述运算重复执行10000次，以使得计时更为准确。最后，我们统计执行一次运算的平均时间。测试结果见表3.1。

表3.1 小规模矩阵测试结果

|  |  |  |  |
| --- | --- | --- | --- |
| IECAMatrix | Matlab | OpenCV | Eigen |
| 4.95\*10-5s | 1.06 \* 10-4s | 9.34\*10-4s | 7.17\*10-5s |

* **大规模矩阵测试**

在大规模矩阵测试中，我们将N设置为3000。这里的重复次数减少为10。测试结果见表3.2。

表3.2 大规模矩阵测试结果

|  |  |  |  |
| --- | --- | --- | --- |
| IECAMatrix | Matlab | OpenCV | Eigen |
| 0.76s | 1.66s | 27.63s | 0.99s |

从上面的两组实验结果可以看出，无论是在大矩阵运算还是在小矩阵运算，IECAMatrix都有着最高的效率，和IECAMatrix效率最接近的是Eigen矩阵运算库。其余两个矩阵库的差距是数量级上的。

在小规模测试中，IECAMatrix的优势更明显。其一是因为，通过IECAMatrix的表达式优化，可以直接将A + B \* C的结果写入A。而事实上，通过剖析Eigen的源码，其做法是将B \* C存入临时变量D，再执行A = A + D。这样做需要频繁开辟新的空间，这导致时间和内存开销均不是最优。

在大规模测试中，上述问题所带来的影响，相比较巨大规模的矩阵乘法所带来的开销而言，就小很多了。当然，这种情况下，IECAMatrix仍然具有很高的效率。

# 第4章 全连接网络的实现

本章介绍了全连接网络的基本结构以及在平台中具体的实现方法。

## 4.1全连接网络结构

神经网络全连接的基本结构见图4.1，这是一个简单的2层神经网络处理二分类问题。它有三维输入，一个3节点的隐含层以及一个单一神经元的输出层。在输入层和隐含层向下一层传递时添加了一个值为1的偏置。二分类问题的输出节点一般采用sigmoid激活函数。

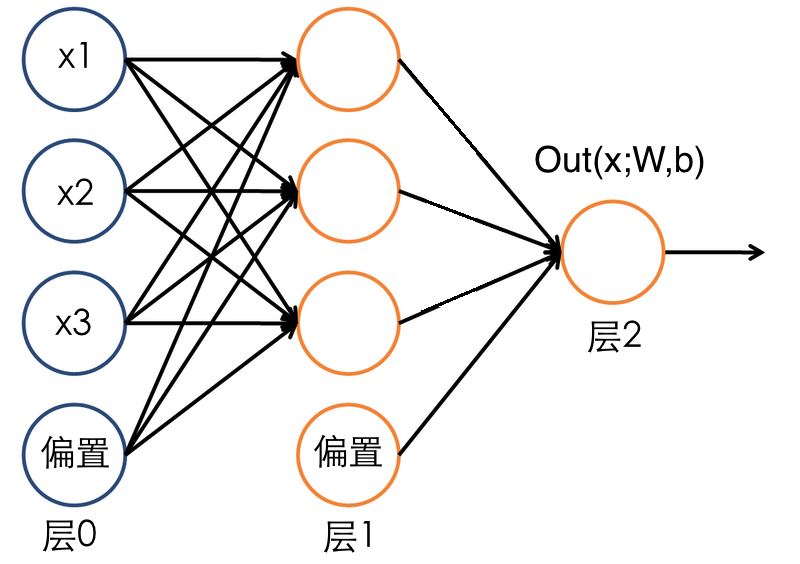


图4.1 神经网络全连接基本结构

## 4.2 全连接网络训练算法[[[22]](#endnote-23)]

求解神经网络分为两步，首先前向传播得到输出结果，之后根据输出值与标记样本计算代价函数，利用批量梯度下降或其他算法求解神经网络。

当前向传播时，每次将 个 维的输入数据合为一批，组成一个 维的输入矩阵，得到输入层的激活值。之后对于除输入层外的每一层进行循环，将上一层得到的激活值与层与层间的权值矩阵做矩阵乘法，得到本层神经元的输出值。对于隐含层，用激活函数(对于本任务，通常使用Sigmoid函数或Rectified函数) 激活本层神经元的输出值，作为本层神经元的激活值。对于输出层，由于是多分类问题，所以需要用SoftMax函数作用在本层神经元的输出值，得到各个分类的概率分布。输出概率最高的神经元可被认为是本样本的输出值。对于二分类问题，可使用单一输出的神经元，输出层的激活函数可以选Sigmoid函数。

在得到神经网络的输出值后，我们可以计算整体的代价函数。之后我们使用批量梯度下降法来求解神经网络，目的是为了得到合适的权重值与偏置值，使得代价函数最小。因为代价函数(本任务中使用均方差函数)是一个非凸函数，故梯度下降法很可能收敛到局部最优解。但是在实际应用中，梯度下降法通常能非常快速的得到一个令人满意的结果。

在梯度下降法中，每一次迭代都要对权重值与偏置值求导，故计算偏导数是求解过程中的关键步骤。在神经网络中，通常使用误差反向传播算法来计算偏导数。反向传播算法的思路如下：给定一个样例 ,我们首先进行“前向传导”运算，利用前向传导公式，得到 直到输出层 的激活值。对输出层（第 \textstyle n_l 层），计算其残差值：

，对于 \textstyle l = n_l-1, n_l-2, n_l-3, \ldots, 2 的各层，计算其残差值：

最后计算最终需要的偏导数值：

神经网络训练算法流程如以下伪代码所示：

**算法：全连接神经网络训练算法**

01 **输入**：样本集 ，网络权重初始化范围，梯度下降学习速率alpha。

02 **输出**：神经网络权重以及网络偏置。样本输出值。

03 初始化网络权重。网络偏置置零。

**第一部分：神经网络训练算法**

04 对于 – 每批m个数据 做

05 前向传播

06 误差反向传播

07 结束 – 循环

**第二部分：前向传播算法**

08 对于 – 每层 神经元 做

09 如果 是输入层 做

10

11 否则 如果 是隐含层 做

12

13

14 否则 如果 是输出层 做

15

16

17 结束 – 如果

18 结束 – 循环

19 对于 – 每个 输出层神经元

20 寻找激活值最大的神经元作为该样本输出

21 结束 – 循环

**第三部分：误差反向传播**

22 对于 – 每层 神经元 做

23 如果 是输出层 做

24 ;

25 否则 如果 是隐含层 做

26

27

28

29

30

31 结束 – 如果

32 结束 – 循环

第5章 word2vec的实现

本章介绍了word2vec实验的基本网络结构以及在平台中具体的实现方法。

## 5.1 人工神经网络词汇表达模型

现阶段人工神经网络词汇表达模型多使用词向量表达，又称分布式词汇表达(Distributed word representation)，指使用低维、稠密的实数向量来表达词汇。词汇的语义信息通过向量各维的数值进行表达。随着互联网上文本数据的爆发性增长，以及深度学习技术的发展，分布式词汇表达广泛被应用在许多的文本挖掘任务中。许多研究者也提出了有效、高效的分布式词汇表达模型和训练算法。

词向量表达通常使用无监督机器学习算法产生，多数算法使用上下文相关的语言模型，以人工神经网络或其变种作为概率模型。无监督机器学习不需要人工对训练数据进行标注，可以充分利用互联网上海量的语料数据进行训练。训练所得的词向量可用于机器翻译、文本挖掘、文档分类等多种领域。

自2003 年Bengio 等人发表人工神经网络概率语言模型以来，出现了众多基于人工神经网络训练词向量表达的方法。2005 年，Morin 和Bengio 发表了基于层次结构的Softmax 模型，使用树状层次结构代替原有的平铺结构，将算法的时间复杂度从O(n)降为O(log(n))，大大提高了算法的效率。2010 年，Mikolov 等人发表了Word2Vec 项目，提出了CBOW（Continuous Bag-of-word，连续词袋）模型，和Skip-Gram 模型（如图5.1）。Word2vec 项目的两种模型对人工神经网络进行了进一步简化，并且吸收了 Morin 的树状层次结构，使得训练效率进一步提高。Mikolov 还提出了一种语义类比问题，用于评价词向量表达的质量。2013 年的NIPS 会议上，Mikolov 等人发表了Word2Vec 项目的改进版本，加入了负样本概率模型，再一次提高了模型的训练效率。

## 5.2 word2vec的基本网络结构

word2vec可用来计算连续分布式词汇表达。它利用文本语料库作为输入数据，格式为词表大小的1-of-v向量表示词汇，输出的模型为低维的词汇表达向量。

word2vec使用两种词汇模型：CBOW模型和Skip-gram模型。如图5.2所示，将输入数据语料库中的文本使用固定大小的滑动窗口进行截取，选取其中词汇，在CBOW模型中，将窗口中心词汇作为标记信息，其余词汇作为训练样本进行输入。相反的，Skip-gram模型中使用中心词汇来预测其上下文中其他词汇。

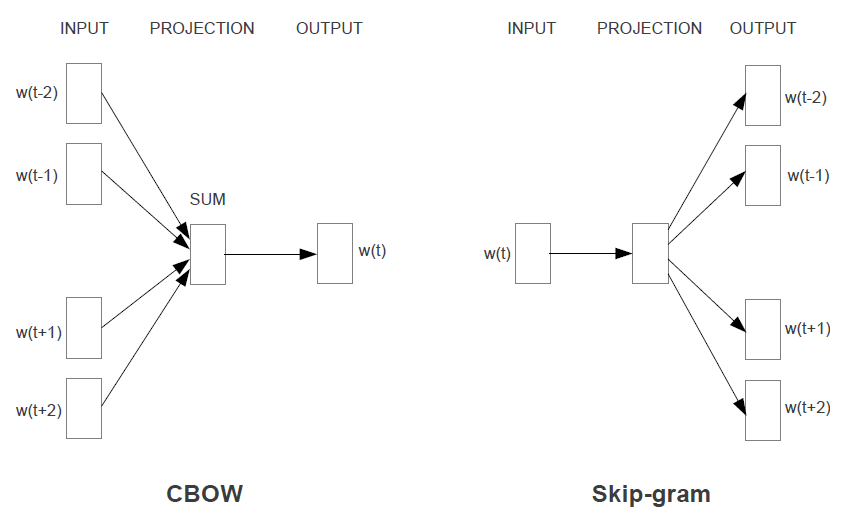


图5.1 CBOW模型和Skip-gram模型

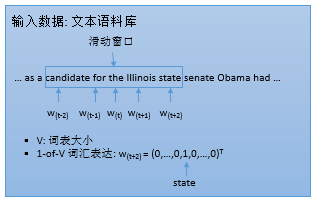
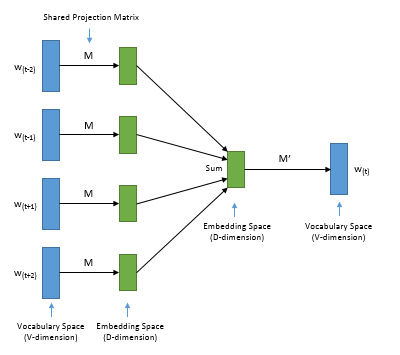
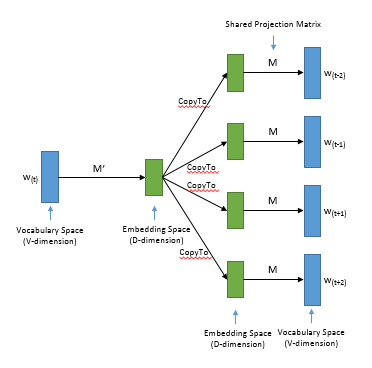


图5.2 滑动窗口取词

如图5.3(a)，CBOW模型是一个两层神经网络结构的模型。输入层维度是词表大小，输入数据为n-of-v的词汇向量，表示窗口中在中心词上下文的n个词。隐含层维度为词汇表达向量空间的维度。输入层与隐含层之间连接的权值矩阵即词汇向量表达矩阵。输出层为1-of-v的预测中心词汇。Skip-gram模型的神经网络结构如图5.3(b)，它利用中心词预测上下文，所以输入层输入1-of-v的中心词汇，而滑动窗口中的其他词汇则在输出层作为标记信息。相应地，隐含层与输出层之间连接的权值矩阵为词汇向量表达矩阵。



1. CBOW模型



1. Skip-gram模型

图5.3 word2vec神经网络结构

* 1. word2vec的实现——softmax近似

在许多分类问题中，常常采用Softmax作为神经网络输出层的激活函数。

对于word2vec的两个模型，其输出层的标记信息为1-of-v或n-of-v的系数向量，而维度v则是词表大小。由于词表大小的数量级往往在百万甚至更多，所以按照普通神经网络的结构对于输出层做softmax操作求输出概率分布，对于word2vec模型来说计算量和时间开销过于昂贵，所以在这里通常采用softmax近似的做法。

### 5.3.1层次Softmax

层次Softmax方法是Softmx近似做法中的一种，它利用哈夫曼编码的方式将计算softmax的方式进行层次化。利用哈夫曼编码可以很好的解决由于词表过大使训练过慢的问题。其基本思想为：利用哈夫曼编码对词表进行压缩，以减小输出层的尺寸。其基本结构见图5.4。

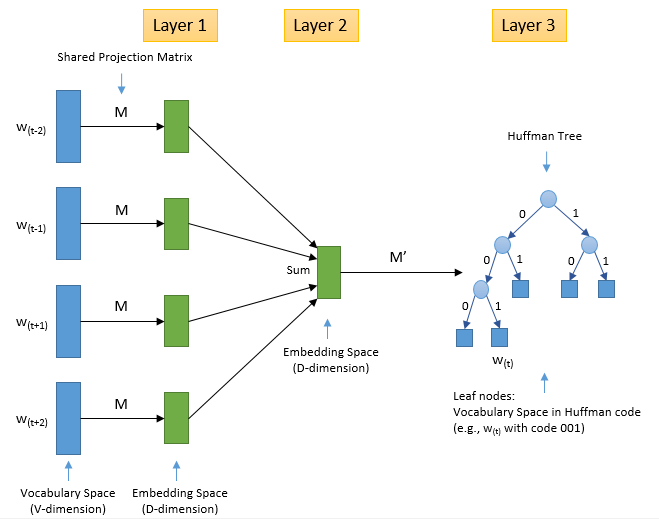


图5.4 利用哈夫曼编码对输出层进行压缩

对于每一个哈夫曼树的节点，都可以将它看作一个二分类器。在整个树路径上节点之间的分类器相互独立的假设下，用或不用哈夫曼编码对于训练来说被证明是等价的。训练的目标是最小化目标函数：

其中与见表5.1。

表5.1 目标函数参数列表

|  |  |  |
| --- | --- | --- |
| **Model** |  |  |
| CBOW | context words | current word |
| Skip-gram | current word | context words |

对于softmax函数，其中：

表示输入词汇向量(M中的一行)，表示输出词汇向量(M’中的一列)。对于这个表达式，需要W次计算操作，开销过大。

而对于经过哈夫曼编码的分层(Hierarchical) softmax函数：

从根走到词的叶节点的概率：

其中 是哈夫曼编码的长度，

经过哈夫曼编码压缩后，只需要次计算即可得到结果。

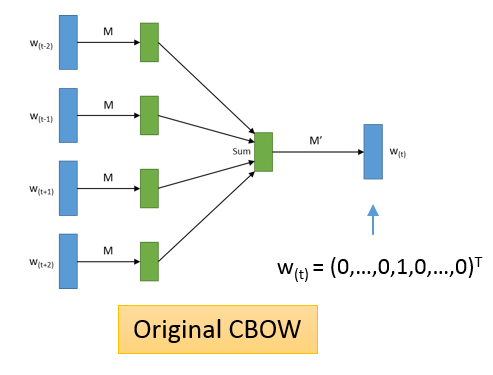
在误差反向传播时，输出层的偏导数为：

### 5.3.2 负样本采样法

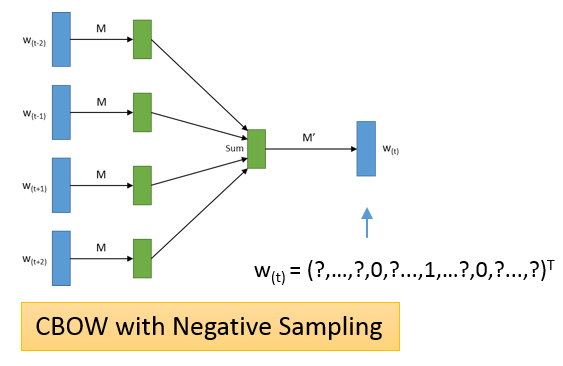
除了哈夫曼编码以外，负样本采样法也是对word2vec的输出层做softmax近似的一种很好的方法。它的基本思想是利用少量的负样本采样以及标记的正样本来计算目标函数以减小计算量。其结构如图5.5(b)所示。

在原始的CBOW模型中，一些相关的词汇(例如同义词)在输出层被视为负样本，可能会有损训练的效果。而对于负样本采样法来说，仅有少量的随机挑选的词在输出层被视为负样本。这样在计算时，可以用以下公式代替：

此时的计算次数由原来的W次降为k+1次(k为所采样的负样本个数)。



1. 原始的CBOW模型网络结构



1. 使用负样本采样的CBOW模型网络结构

图5.5 原始模型与负样本采样结构对比

第6章 卷积神经网络的实现

本章介绍了卷积神经网络的基本结构以及在平台中具体的实现方法。

* 1. 卷积神经网络的基本结构

在 2006 年深度置信网的训练算法[[[23]](#endnote-24)]提出以前，基于生物学研究成果，由Fukushima、LeCun等人提出并发展的卷积神经网络是为数不多的，可以在避免过拟合情况下，进行多隐层训练的神经网络，具有结构简单、训练参数少和适应性强等特点。特别是在图像识别领域，可以直接对二维图像（而无需一维化）进行建模，对平移、比例缩放、倾斜或者共他形式的变形具有高度不变性。

LeCun在1998 年提出了首个进行了实际应用的神经网络模型：LeNet-5。卷积神经网络由于其权值共享、参数自由度较小的特点，深层训练时，不易发生过拟合。特别是其卷积的特性使得其在处理二维图像上有着很强的优势。

下面简单介绍LeNet-5的基本结构：

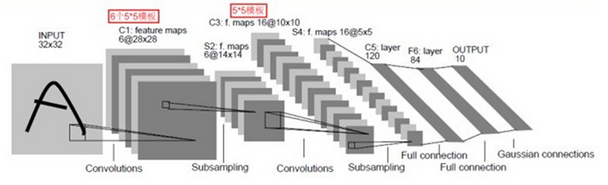


图6.1 LeNet-5的基本结构

如图6.1所示，LeNet-5共有7层，它将MNIST数据库的每张图片进行填充(padding)后作为输入。MNIST手写数据库的图片大小为28\*28，填充操作将每张图片外围填补2层零值，即原图变为32\*32大小。填充操作的目的是希望图片边缘的明显特征能够出现在最高层特征监测子感受野(卷积窗口)的中心。

第一层为卷积层，卷积模板大小为5\*5，步长为1，共使用6个可训练的卷积模板。使用卷积模板对输入图像进行卷积过后，加一个偏置b，得到第一个卷积层。第一个卷积层有6张卷积过后的特征图片，大小为28\*28。卷积层的目的是提取图片特征。

第二层为池化层，池化窗口为2\*2，步长为2。池化操作即对池化窗口内的像素点取最大值或平均值作为下一层图片对应位置的值，称之为最大值池化或平均值池化。池化后的图片大小为14\*14。池化后再经过激活函数，即得到池化层。池化层的目的是增加神经网络对图片细微旋转和偏移的鲁棒性。一次卷积和池化的完整操作见图6.2的。

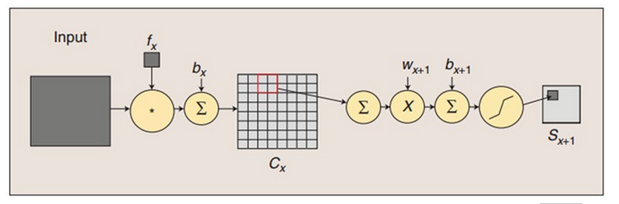


图6.2 卷积与池化操作

第三层依然是卷积层，拥有16个大小为5\*5,步长为1的卷积模板，卷积后得到16张大小为10\*10的特征图片。本层得到的特征图片表示的是上一层提取到特征图片的不同组合(这里类比人的视觉系统，底层的结构构成上层更抽象的结构，例如边缘构成形状)。第四层是池化层，池化窗口

为2\*2，步长为2。第五、六、七层均为全连接层，分别拥有120, 84, 10个神经元，其中第七层为输出层。

* 1. 卷积神经网络的实现

由于图像的存储与输入格式采用一维数组的方式，在进行卷积操作时，寻址的开销与计算的开销非常昂贵。为了避免这种情况，考虑到任务的实际情况进行相应优化。

对于输入图像，由于有多个相同大小的卷积模板要对其进行卷积，所以考虑事先将输入图像按照卷积相乘时所对应的位置将其展开。我们将这种操作叫做im2col。当对输入图像进行im2col操作后，得到展开后的图像，在做前向传播时即可直接与权值矩阵相乘，得到原图与权值矩阵进行卷积后的效果。在反向传播时，直接将误差信号与权值矩阵相乘，得到被展开的误差信号，此时做im2col的反向操作，将误差信号完整的传播到下一层神经元上。

具体的流程如下：

* 1. 卷积连接中的前向传播。

1. 前层神经元的输入：使用I.N×I.C×I.H×I.W长度的向量表示输入图像，I.N表示批处理图像时一次训练的样本数量， I.C表示前层卷积特征图像的个数（对于输入层即输入图像通道数），I.H表示输入特征图像的高度， I.W表示输入特征图像的宽度。
2. 卷积连接的权值矩阵：F.N×F.C×F.S×F.S，F.N表示本层卷积操作所需卷积模板的个数(即生成特征图像个数)， F.C=I.C，表示前层卷积特征图像的个数，F.S×F.S是卷积窗口的大小。
3. 后层神经元的输出：O.N×O.C×O.H×O.W，O.N=I.N，表示批处理图像时一次训练的样本数量，O.C=F.N，表示输出特征图像的个数（也即本层卷积连接的卷积模板个数），O.H×O.W表示输出图像的大小。
4. 依次对每张输入图片进行采样展开(im2col)：如图6.3，令大小为F.S×F.S的卷积窗口在大小为I.H×I.W，通道数为I.C的输入图像上滑动，将每次采样得到的F.C×F.S×F.S个数值依次填入Col\_data矩阵的一列。采样完毕后，可得到将输入图像按照卷积顺序展开的完整Col\_data矩阵。



图6.3 Im2col操作

1. 将权值矩阵二维化，使其行向量表示一个卷积模板的所有权重，行数表示本层卷积模板的数量。
2. 如图6.4，计算后层神经元的输出，将权值矩阵与展开后的输入图像相乘，得到输出的特征图像。



图6.4 前向传播

二、卷积连接中的误差反向传播

1. 误差反向传播时的误差信号维度与输入图像相同。

2. 首先传播误差信号，按照公式，如图6.5，计算前层神经元采样展开后的误差输出。



图6.5 反向传播误差信号

3. 对于计算所得前层误差输出，需进行反采样展开(col2im)得到前层神经元对应的结果再输出。如图6.6所示。



图6.6 col2im

4. 利用后层神经元误差信号与前层神经元前向传播时的激活值对权重进行更新。首先对前层神经元激活值进行采样展开，得到展开后矩阵。如图6.3。

5. 如图6.7，根据公式 ，对权值矩阵进行更新。



图6.7 反向传播更新权重

第7章 分布式网络的实现

在这一章，我们着重介绍IECABrain在应对大规模机器学习问题时候所采用的算法。主体内容将分为两个部分：1）如何同时用多份训练数据训练同一个模型，来加速对数据集的迭代；2）如何将模型分布式存储，以应对大规模神经网络模型。

除了神经网络本身的算法，我们还将讨论配套的相关技术。例如，在分布式计算中，我们使用MPI（Message Passing Interface）框架；我们的流式数据读取方式如何在多模型和分布式情况下使用。

## 7.1 多模型并发

在上文中，我们介绍了批量梯度算法，用来训练神经网络。由于数据是按照流的形式输入到神经网络中，也就是说，一个批次的数据对应了一次更新。所以一个很自然的想法——我们可以通过并行化这些更新，来减少整个训练时间。

### 7.1.1 模型拷贝

在IECABrain中，我们实现了多线程的多模型训练。如图7.1展示了该训练框架。

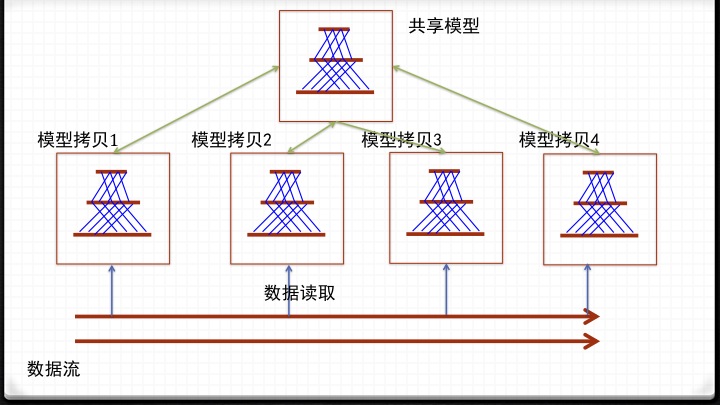


图7.1 模型拷贝训练框架

图中，每个模型拷贝都由一个线程（或者是进程）维护。每个模型拷贝都是一个完整的神经网络。这些模型拷贝是从同一个模型复制而来——即共享模型。每个模型拷贝都会从数据流中读入数据，因此，在上图中，整个训练数据集实际上有四个通道流出，平均而言，每个模型拷贝会用到整个训练数据集四分之一的数据来训练自身。

和仅仅用四分之一的训练数据训练一个神经网络模型不同，上面的四个模型拷贝，时刻保持同步。绿色箭头代表了模型拷贝之间会进行同步。因此，将整个训练数据训练完成后，期望的结果是和原本单独一个模型训练的效果等价。

### 7.1.2 无锁更新

上述模型拷贝如果存在于不同的进程中，那么这些模型拷贝之间需要相互的进程之间的通信。如果存在于不同的线程，那么也需要相应的通信代价。

有通信就会带来额外的代价，而通信代价过高的时候，甚至效率不如单一模型，所以得不偿失。为了最大限度地提升效率，减少通信代价，IECABrain采用了一种多线程无锁异步更新的方式，如图7.2所示：

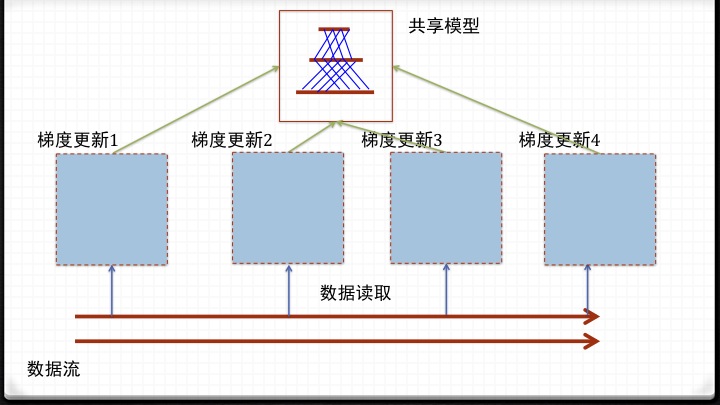


图7.2 无锁更新模型

可以看到，与上图相比，主要的变化有：

1. 没有了模型拷贝。由于是多线程训练，所有的线程共享一份模型数据（即共享神经网络模型的权重和偏置）。
2. 仍然是多线程训练，不过每次训练的结果——即对权重和偏置的梯度更新，将会直接更新到共享模型上。

这样的方法主要的好处有：1) 没有冗余的模型拷贝，节省内存；2) 多线程训练的结果将直接反应到全局的模型参数中，没有额外的通信代价。

所以，在程序效率上，这种方法显然是更为优秀的。不过也会有相应的代价。1) 首先，多线程的无锁更新会带来读写冲突。然而，这种冲突会造成影响的情况，也只是在多个线程同时更新共享模型的时候，所以一种方式，可以允许并发读、顺序写，另外一种方式就是直接忽略这个问题——因为写冲突导致的结果，只是会丢失一次更新，这是可以容忍的；2) 其次，每个训练线程在做前向传播和反向传播之间，模型参数就有可能被其他线程所更新，导致前后不一致。然而，这种问题所导致的结果也要视情况而言。大多数问题中，上述问题并不会使得训练结果在精度上有明显的损失。

### 7.1.3 流式数据提供

在前文中，我们介绍了IECABrain的流式数据提供方CDataSupplier。这里将阐述如何为多个线程提供同一个数据流，并且在保证多线程读取的情况下，保证数据的顺序和避免数据不一致。

在下面的例子中，假设在多模型训练中，开启三个线程进行训练。另外，由于神经网络只要满足有向无环的条件即可训练，所以，我们也必须支持存在多个输入层的情况（例如多任务学习，协同训练等任务）。

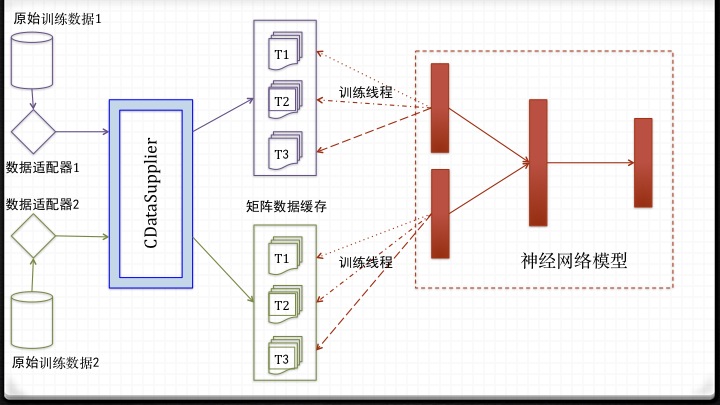


图7.3 多模型数据供应

如图7.3，描述了在多模型的情况下，流式数据如何提供服务，具体如下：

1. 两个原始训练数据文件，对应了神经网络的两个输入层。因而，需要两个数据适配器从原始文件中读入数据（因为这两个原始数据可以是完全不同的格式，如音频、图像）。
2. CDataSupplier会将两个适配器读入的数据分配到两个不同的缓存仓库。每个缓存仓库对应了一个原始数据文件。同时，每个缓存仓库缓存的数据份数正好是训练线程的数量（多模型训练的模型数量）。另外，这里的缓存已经是处理好的矩阵数据，即可以直接放入神经网络中训练，不需要额外的格式转换。
3. 在训练的过程中，每个线程各自从CDataSupplier的对应缓存仓库读取已经准备好格式的矩阵（数据）。当缓存被清空的时候，CDataSupplier会自动调用数据适配器，装入下一个批次的数据。如此反复，直到所有的数据文件都读入到末尾。

## 7.2 分布式训练

在拥有多于一个运算节点（服务器或者PC）的时候，IECABrain能够同时利用多个运算资源，从而进一步提升运算性能，以及能够处理的问题的规模。这一节，我们来关注IECABrain的分布式计算能力。

### 7.2.1 数据分布式与模型分布式

具体而言，IECABrain的分布式体现在如下两个方面：

1. **数据分布式**

在拥有多个运算节点的时候，每个节点可以独自用一部分数据训练一个神经网络模型。只需要在合适的时候，将这些节点之间训练的结果进行同步，最终就能够得到与单个运算节点等价的训练结果。

由于流式数据同时被多个运算节点所需求，因此，这大大加快了训练一整个数据集的效率。如果除去运算节点之间的通信代价，其所带来的扩展能力是相当可观的。即，只要有充足的运算节点，便可以让训练效率进一步提升——这便是上一节提到的多模型训练。在下文中，我们便不再赘述这种分布式的做法了。

1. **模型分布式**

所谓模型分布式，就是将一个神经网络模型拆分成若干部分，每个部分存放在一个运算节点上。如图7.4便是一个直观的例子：

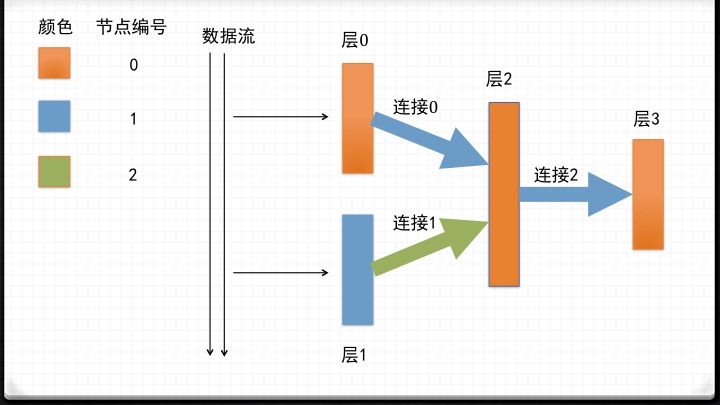


图7.4 分布式网络结构示意

图7.4中展示了一个具有4个层，3个连接的神经网络，网络的拓扑深度是3。由于神经网络的主要内存开销主要来自于层中的神经元激活值存储，以及连接中的边和偏置的权值存储，因此，层和连接都可以位于不同的计算节点中。图中一共有四个运算节点，分别用不同颜色来表示。相同颜色的层和连接代表它们存在于同一个运算节点。

以上的模型分布式使得IECABrain具有如下的能力：

* 1. **训练大规模神经网络**

在一些需求中，神经网络的输入层可能非常大（例如，数十亿维度的图像数据或者搜索广告数据）。如果隐层维度也高的话，那么这两层之间的全连接的边数规模就会非常大，导致无法在一台机器上的内存中完整存储（如果用虚拟内存，效率会有数量级的差别，也不可取）。而如果能够将模型进行切分，进而将其存放在不同的运算节点上，那么单个节点的内存压力就会大大降低，同时，我们付出的代价仅仅是多个节点之间的通信开销。在具有InfiniBand连接的服务器集群中，这个开销非常小，与高昂的计算时间相比可以忽略。

* 1. **分担节点计算量，加速神经网络训练**

在上图中可以看出，层0和层1的前向传播和反向传播之间没有依赖关系，因此可以并行计算。又由于层0和层1存在于不同的运算节点中，因此，其使用的计算资源也不冲突。通过并行计算，在分担运算节点的计算量的同时，也加速了神经网络的训练。

下文将着重阐述IECABrain是如何实现模型分布式的。

### 7.2.2 模型分布式的实现

在分布式计算的实现中，有两种主流的方式。其一是MapReduce，其二则是MPI。下面对比这两种方式，以阐述我们选择MPI作为分布式计算实现的动机。

* **MapReduce**

该框架通过map和reduce这两种抽象操作，在计算机集群上实现了分布式的计算。Map操作将运算分布式，reduce操作再将具有相同键值的数据归约到同一计算节点进行归约操作。而MapReduce有两个主要的问题，不适合用于神经网络计算中。一方面，神经网络优化采用的是迭代式的优化算法，而MapReduce并不适合；另外一方面，常见的MapReduce服务将数据存放于硬盘，而对于时效性较高的神经网络训练任务，这种做法效率偏低。

* **MPI**

MPI是个消息传递接口（Message Passing Interface），通过统一的接口，将常用的进程间通信封装。用户不需要关注具体的实现（并且事实上，有多套MPI的实现，各有优劣），只需遵循MPI接口。另外，无论多个进程是存在于同一个节点，还是存在于不同的节点，对于MPI程序没有区别。MPI程序将数据存于内存中，因而效率较高。综合以上的原因，我们采用MPI作为分布式计算的底层通信依赖。

现在，我们以上图中的网络分割方案为例，阐述分布式实现方法。

为了让运算节点之间协同工作，每个运算节点都将会知道完整的网络结构（但是存储只在必须的运算节点上完成）。如图7.5所示：

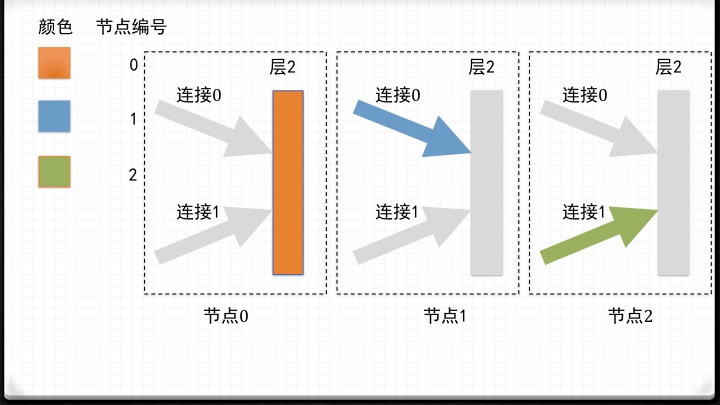


图7.5 各节点存储情况示意图

图7.5展示了进行层2的前向传播时候，各个节点上的存储情况。

由于层2存放于节点0，而前向传播所需要的连接0和连接1都在别的节点（节点1和节点2）上，所以，为了让节点0上具有实际应该有的前向传播的激活值，需要让节点1和节点2各自计算出来的激活值传递给节点0。

我们把上面的过程叫做合并激活值。

当激活值合并以后，层2就可以对其进行非线性变换（例如使用Sigmoid函数，或者Tanh函数）。为了能够将层2的状态继续向前传播，我们需要将层2的状态进行分发，如图7.6所示：

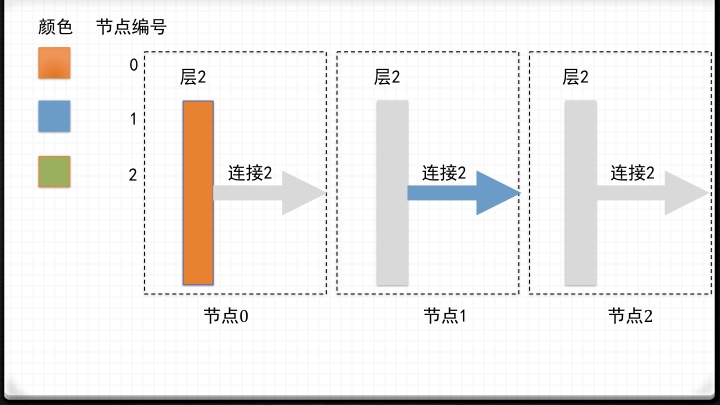


图7.6 分发状态示意图

由于在之后的前向传播过程中，只有连接2需要用到层2的状态，因此，存放于节点0的层2需要将其状态传输给存放连接2的节点1。由于层2和连接2都与节点2无关，所以在这一局部的前向传播过程中，不需要将数据与节点2进行交换。

我们把上面的过程叫做分发激活值。

反向传播的过程与前向传播类似，我们就不再赘述其分布式算法了。

综上，我们将前向传播和反向传播的伪代码总结如下：

**算法：分布式网络训练算法**

01 **输入**：样本集 ，网络权重初始化范围，梯度下降学习速率alpha，网络分割方式。

02 **输出**：神经网络权重以及网络偏置。样本输出值。

03 初始化网络权重。网络偏置置零。

**第一部分：前向传播**

04 对于 – 每个传播到当前层的连接

05 调用该连接的前向传播

06 结束 – 循环

07 合并每个节点上计算的当前层的激活值

08 计算非线性激活值

09 将当前层的激活值分发到相关的运算节点

**第二部分：反向传播**

10 对于 – 每个当前层的后继连接

11 调用该连接的反向传播

12 结束 – 循环

13 合并每个节点上计算的当前层的累积误差

14 根据当前层的激活函数计算其偏导数

15 将当前层的偏导数分发到相关运算节点

第8章 实验结果分析与对比

本文使用文中所介绍神经网络平台，以及一些通用数据集，对前文所介绍的各部分网络在Windows操作系统下进行了测试。硬件的硬件设备基本信息：CPU版本的测试在型号为Intel Xeon E5-1620 v2, 主频为3.7GHz的8核CPU，内存为16GB的工作站上进行，GPU版本的测试在服务器上进行，显卡型号为：NVIDIA Tesla K20x，单精度浮点运算峰值为3.95Tflps，显存带宽250GB/s，显存大小为6GB，流处理器数量为2688。

* 1. MNIST标准数据集实验

MNIST (LeCun等人, 1998) 数据集由像素灰度的0-9手写数字图像组成，共包括60,000个训练样本以及10,000个测试样本。它是NIST数据库的一个子集，图像已被规范化。MNIST每张图片可以使用一个784维度的向量表征，对于神经网络平台是一个很好的性能测试集。

对于MNIST数据集，我们在不同的神经网络平台上训练了相同的网络结构，以测试各个神经网络平台的性能。我们采用全连接网络的方式进行训练，基本的网络结构如图8.1。该网络为三层神经网络，输入层大小为784维，两个隐含层大小为1024维，输出层目标为10分类。每层之间采用全连接，使用偏置。

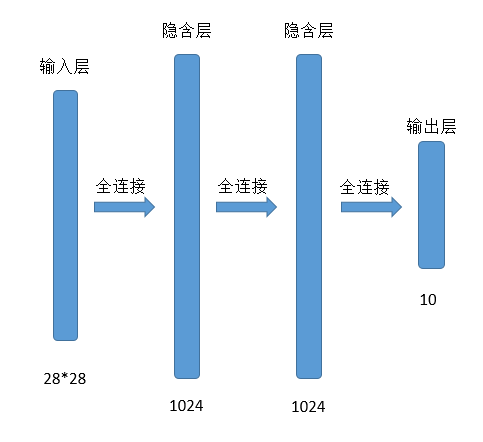


图8.1 MNIST全连接网络结构

由于采用批训练梯度下降方法进行训练，每批数据大小为100。隐含层激活函数采用Rectifier函数，输出层使用Softmax函数。目标函数为Logloss函数。学习速率为0.01。

首先选择在CPU平台进行测试，训练迭代次数为1，训练结束后，性能测试结果见表8.1。由于各个神经网络平台所采用的网络结构与网络配置相同，所以其测试准确率结果相同，在此不再赘述。

表8.1: MNIST标准数据集在CPU不同神经网络平台下性能测试

|  |  |
| --- | --- |
| Matlab Deeplearning Toolbox | 33.61s |
| Opencv | 347.56s |
| Caffe | 10.66s |
| IECABrain + 1 thread | 7.31s |
| IECABrain + 3 threads | 4.03s |

由表8.1中可见，本文所述平台在此任务中，速度明显优于其他主流平台。

其中caffe平台与本文所述平台在底层函数的调用相同，故时间差距较小。Caffe平台支持在GPU下的神经网络训练，故选用该平台在GPU下做进一步性能测试。网络结构与配置与CPU下测试相同，迭代次数为1，测试结果如表8.2。

表8.2: MNIST标准数据集在GPU不同神经网络平台下性能测试

|  |  |
| --- | --- |
| Caffe | 1.65s |
| IECABrain | 1.50s |

由表8.2可见，在GPU平台下，本文所述平台速度要优于caffe平台。另对比CPU下测试结果可知，MNIST任务在GPU平台下的加速比可达5倍以上。

* 1. 特殊网络实验

### 8.2.1 卷积神经网络实验

如6.1节所述，LeNet-5卷积神经网络在MNIST数据集上取得了很好的结果，可以利用该数据集作为本文所述平台卷积连接的测试数据集。

对于MNIST数据集，我们在不同的平台上搭建了相同的网络结构与配置，以测试各个平台卷积连接的性能。基本网络结构见表8.3：首先对输入层的单通道图像进行卷积操作，采用6个大小的卷积窗口，卷积步长为1。在卷积层后使用Rectifier函数进行激活，激活过后进行池化操作，选择平均值池化函数，池化窗口大小为，得到大小为的第一层特征图像。之后再进行卷积操作，采用16个大小的卷积窗口，卷积步长为1。在卷积层后使用Rectifier函数进行激活，激活过后进行池化操作，选择最大值池化函数，池化窗口大小为，得到大小为的第二层特征图像。之后采取两层全连接网络，分别连接到120、80个神经元，最后使用Softmax函数进行输出。

表8.3: LeNet-5卷积神经网络结构

|  |  |  |
| --- | --- | --- |
| 操作 | 参数 | 操作后图片通道及大小 |
| 输入 |  | 1\*28\*28 |
| 卷积 | 6\*5\*5 | 6\*24\*24 |
| 非线性激活 | Rectifier |  |
| 平均值池化 | 2\*2 | 6\*12\*12 |
| 卷积 | 16\*5\*5 | 16\*8\*8 |
| 非线性激活 | Rectifier |  |
| 最大值池化 | 2\*2 | 16\*4\*4 |
| 全连接 | Sigmoid | 120 |
| 全连接 | Sigmoid | 84 |
| 输出 | Softmax | 10 |

由于采用批训练梯度下降方法进行训练，每批数据大小为100。目标函数为Logloss函数。学习速率为0.005。在CPU平台下，使用MNIST数据集以及上述网络结构对各平台进行性能测试。测试迭代次数为1，测试结果见表8.4。

表8.4: CIFAR-10数据集在CPU不同神经网络平台下性能测试

|  |  |
| --- | --- |
| Matlab Deeplearning Toolbox | 78.53s |
| Caffe | 13.86s |
| IECABrain + 5threads | 7.10s |

由表8.4可见，本文所述平台在此任务中，速度明显优于其他测试平台。

### 8.2.2 word2vec实验

Text8数据集是Mikolov的word2vec的demo训练数据，是Enwiki9数据集的一部分。Enwiki9是英文版维基百科的全部语料库，其中包含约123.4M个训练单词，词表大小约为220K左右。Enwiki9是文本学习领域重要的语料库，可以用作word2vec工具的训练样本集。

我们利用本文所述平台对Text8数据集进行了训练。训练使用CBOW模型并对输出层使用负样本采样法进行softmax近似。窗口大小为5，测试迭代次数为1，测试结果见表8.5。

表8.5: Text8数据集在CPU下平台性能测试

|  |  |
| --- | --- |
| IECABrain + 8 threads | 20.14s |

* 1. 分布式实验

如上文介绍本平台分布式网络的实现，可以将大规模网络或数据集切分到多台主机端进行运行。现用上文所述数据集测试分布式网络实现的性能。

### 8.3.1 MNIST数据集

在MNIST数据集上，将本章第一节中的网络结构进行切分。如图10所示，将输入层784维切分为2个392维的输入层，将隐含层1024维切分成2个512维的隐含层。总共将该网络切分成7层，并随机分配到4台服务器上进行训练，具体分配见表8.6。

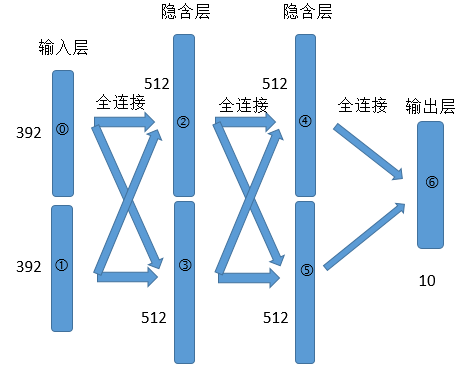


图8.2 分布式MNIST全连接网络

表8.6: 分布式MNIST网络与服务器分配

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 层编号 | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| 服务器编号 | 2 | 3 | 0 | 2 | 1 | 3 | 0 |

网络基本配置同本章第一节中实验，测试迭代次数为1，测试结果见表8.7。

表8.7: MNIST分布式实验

|  |  |
| --- | --- |
| IECABrain | 10.00s |

### 8.3.2 Text8数据集

在word2vec实验中也可进行分布式实验，当数据集过大，导致词表长度过大、输入输出层已经无法被储存在单台服务器的内存或单块显卡的显存中时，可以将输入输出层进行切分，放到多台服务器中，进行大规模分布式文本的学习与训练。

在Text8数据集上，将本章第二节的网络结构的输入层平分为2层，分别放到两台不同的服务器上，输出层同样进行切分，放到两台不同的服务器上，以此结构进行分布式网络的训练。

网络基本配置同本章第二节中的实验，测试迭代次数为1，测试结果见表8.8。

表8.8: Text8分布式实验

|  |  |
| --- | --- |
| IECABrain | 317.77s |

由此可见，本文所述神经网络平台对用于大规模文本学习的分布式网络的训练支持良好。

# 第9章 总结与展望

## 9.1 文章小结

本文设计并实现了一种快速、灵活、可支持大规模训练的神经网络平台。其中包含矩阵库部分与神经网络框架部分。

矩阵库的底层函数调用选择了MKL库与CuBlas库进行加速，上层封装使用表达式模板类的方法，优化表 达式，减少计算量以及额外空间的消耗。实测结果性能优于Eigen矩阵库。

神经网络框架部分分别实现了全连接神经网络、卷积神经网络、word2vec工具模型以及分布式的网络。并在CPU及GPU平台分别作了相关性能测试。

## 9.2 未来工作展望

本文所述平台可扩展性强，具有非常大的继续开发潜力。目前平台中关于神经网络的一些常用技巧还并不完善，例如随机舍弃(DropOut)，本地规范化(Local normalization)等。

# 参考文献

1. [] Rumelhart D E. Parallel distributed processing[M]. IEEE, 1988. [↑](#endnote-ref-2)
2. [] Pomerleau, Dean A. "Efficient training of artificial neural networks for autonomous navigation." *Neural Computation* 3.1 (1991): 88-97 [↑](#endnote-ref-3)
3. [] LeCun, Yann, et al. "Gradient-based learning applied to document recognition." *Proceedings of the IEEE* 86.11 (1998): 2278-2324. [↑](#endnote-ref-4)
4. [] Hinton G E, Salakhutdinov R R. Reducing the dimensionality of data with neural networks[J]. Science, 2006, 313(5786): 504-507. [↑](#endnote-ref-5)
5. [] Hinton G E, Srivastava N, Krizhevsky A, et al. Improving neural networks by preventing co-adaptation of feature detectors[J]. arXiv preprint arXiv:1207.0580, 2012. [↑](#endnote-ref-6)
6. [] Glorot X, Bordes A, Bengio Y. Deep sparse rectifier networks[C]//Proceedings of the 14th International Conference on Artificial Intelligence and Statistics. JMLR W&CP Volume. 2011, 15: 315-323. [↑](#endnote-ref-7)
7. [] Raina R, Madhavan A, Ng A Y. Large-scale deep unsupervised learning using graphics processors[C]//ICML. 2009, 9: 873-880. [↑](#endnote-ref-8)
8. [] Wan L, Zeiler M, Zhang S, et al. Regularization of neural networks using dropconnect[C]//Proceedings of the 30th International Conference on Machine Learning (ICML-13). 2013: 1058-1066. [↑](#endnote-ref-9)
9. [] Mikolov T, Chen K, Corrado G, et al. Efficient estimation of word representations in vector space[J]. arXiv preprint arXiv:1301.3781, 2013. [↑](#endnote-ref-10)
10. [] Theano. <http://deeplearning.net/software/theano/> [↑](#endnote-ref-11)
11. [] Deep Learning Toolbox. <http://www.mathworks.cn/matlabcentral/fileexchange/38310-deep-learning-toolbox> [↑](#endnote-ref-12)
12. [] Cuda-convnet. <https://code.google.com/p/cuda-convnet/> [↑](#endnote-ref-13)
13. [] Caffe. <http://caffe.berkeleyvision.org/> [↑](#endnote-ref-14)
14. [] Accord.NET . <https://code.google.com/p/accord/> [↑](#endnote-ref-15)
15. [] MATLAB. <http://www.mathworks.cn/products/matlab/> [↑](#endnote-ref-16)
16. [] PyBrain. <http://www.pybrain.org/> [↑](#endnote-ref-17)
17. [] FANN. <http://leenissen.dk/fann/html/files2/installation-txt.html> [↑](#endnote-ref-18)
18. [] OpenCv. <http://opencv.org/> [↑](#endnote-ref-19)
19. [] Eigen. <http://eigen.tuxfamily.org/> [↑](#endnote-ref-20)
20. [] Intel M K L. Intel Math Kernel Library[J]. 2007. [↑](#endnote-ref-21)
21. [] Nvidia C. Programming guide[J]. 2008. [↑](#endnote-ref-22)
22. [] Ng A, Ngiam J, Foo C Y, et al. UFLDL tutorial[J]. 2012. [↑](#endnote-ref-23)
23. [] Hinton G E, Osindero S, Teh Y W. A fast learning algorithm for deep belief nets[J]. Neural computation, 2006, 18(7): 1527-1554. [↑](#endnote-ref-24)